Making Beautiful Graphs with Zplot

Remzi H. Arpaci-Dusseau
University of Wisconsin, Madison

Abstract

This paper introduces Zplot, a Tcl library for making two-dimensional data plots. Zplot provides a simple set of primitives that allow users to input and manipulate data, plot said data in a variety of formats, and decorate the resulting graphs with axes, labels, and other textual accents. Zplot then outputs encapsulated PostScript for ease of inclusion in technical documents.

1 Introduction

Over the past 20 years or so, I have used a variety of tools to generate data graphics for the various technical papers with which I have been involved. These tools left me despondent. They seemed incapable of producing all but the most basic of graphs. Many common graph types were not well supported (e.g., bar graphs). Simple data manipulations were forced into pre-processing steps, creating a clumsy tool chain. Manual manipulation on the resultant PostScript was often required to achieve the desired result.

Zplot is the fruit that was born of this frustration. Zplot is a pure Tcl library that allows the creation of two-dimensional data graphics in a flexible and powerful manner. Typical graphs are created with only a few lines of Tcl, and complex and intricate graphs can be produced from only tens of lines of code.

In this document, I describe Zplot. First, I give an overview of the tool and the basic primitives it provides. Then, I describe each of the basic routines in more detail, showing how they can be combined to produce a wide range of interesting graphs. Zplot drawing routines are all built upon a set of low-level PostScript-generating commands; these hide many of the details of generating correct PostScript from the rest of Zplot, boiling down most activities to simple drawing commands that place lines, shapes, and text on the drawing surface. I then conclude the paper with a few comments about Tcl, related and future work, and a final summary.

2 Overview

I now describe the basic primitives provided by Zplot. Let us start with a typical (if simple) graph as an example, and use this to drive the discussion of the different elements of Zplot. A typical graphing script might be written as follows (with the results of the graph shown in Figure 1).

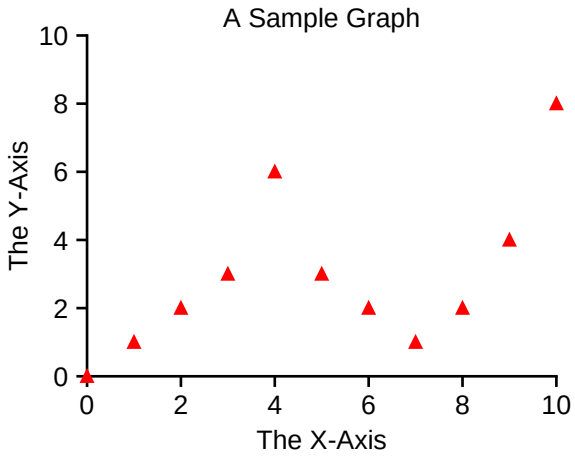

Figure 1: **An Example Graph.** *The most bare-boned of plots that one can make with Zplot.*

```
# input the library
source zplot.tcl
namespace import Zplot::*
# describe the drawing surface
PsCanvas -title "file.eps" -dimensions 300,200
# load some data
Table -table t -file "file.data"
# make a drawable region for a graph
Drawable -xrange 0,10 -yrange 0,20
# make some axes
AxesTicsLabels -title "A Sample Graph" \
  -xtitle "The X-Axis" -ytitle "The Y-Axis"
# plot the points
PlotPoints -table t -xfield x -yfield y \
  -style triangle -linecolor red
# finally, output the graph to a file
PsRender -file "file.eps"
```

In this example, the user creates a graph by first describing the drawing surface by calling `PsCanvas` and specifying its dimensions. Then, the user calls the `Table` routine to load data into Zplot, getting the data from a file (`file.data`). The user, wishing to plot the data, now creates a drawable region by calling the `Drawable` routine; doing so defines where on the canvas the drawable is, and also how to map data points onto the drawing surface (e.g., the range of x values and y values that map onto this drawable). With a drawable defined, the user can now call

one of a variety of plotting routines (e.g., `PlotPoints`) to plot the data onto the drawable. The plotting routines generally take a large number of arguments, enabling a wide variety of plots to be produced; in this case, the user chooses to draw a red triangle at each (x,y) point of the graph. Finally, the user adds some graphical and textual decorations to help clarify the graph (in this case, by simply calling the `AxesTicsLabels` routine), and then renders the PostScript to a file by calling PsRender. I now describe each of these primitives in more detail.

Note that each of these routines takes a large number of optional parameters. To find out what these are (without perusing the source code), one should simply call the routine and pass it the `-help` flag (or any bad flag); a useful error message about the routine and all of its parameters (including default values) will be printed.

2.1 Table

There are numerous routines available to users to input and manipulate data; these are known as the `Table*` routines. The most commonly used routine is the basic `Table` routine; usually, this routine is used to input a file and then plot its points. A typical file (such as `file.data` above) looks like this:

```
# x y
0 0
1 1
2 2
3 3
4 6
...
9 4
10 8
```

The first line contains the "schema" for the table, with names for each column; these names are subsequently used to refer to the data when manipulating it or drawing it to the screen.

One powerful routine is the `TableSelect` command; it allows one to perform a database-like selection over a table and put the results in a new table. Here is an example that selects data from table `t` with y-values above 5, and plots green circles around said points (the results of which are shown in Figure 2):

```
Table -table thi -columns x,y
TableSelect -from t -to thi -where {$y > 5}
PlotPoints -table thi -xfield x -yfield y \
  -style circle -linecolor green -size 4
```

There are a number of other useful table functions which are not covered here, mostly for manipulating and summarizing data. For example, `TableMath` can be used to perform a mathematical operation (or indeed, any valid Tcl expression) on a column of data. The routine `TableComputeMeanEtc` is useful for computing means and deviations over a column, and `TableBucketize` can be used to place data into bins. All of these primitives are built on lower-level table routines that access each row of a table and perform operations on its contents; thus, more complex operations on tables can be readily assembled by adventurous users.

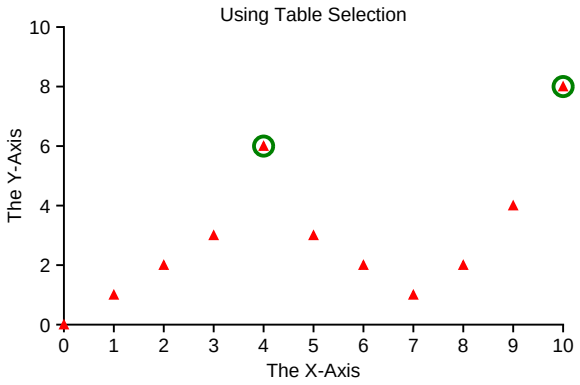

Figure 2: **Table Selection.** *The example uses a simple table selection to find y-values that are greater than 5. Then, these points are plotted as green circles.*

2.2 Drawable

The drawable is likely the most important abstraction that Zplot implements. A drawable is created by the `Drawable` command. Each drawable has a name; the default name is `default` and this default is used by all routines that expect a drawable unless otherwise specified. Here is the relevant portion of the example above rewritten to use the drawable name `foo` instead of the default:

```
Drawable -drawable foo -xrange 0,10 \
  -yrange 0,20
AxesTicsLabels -drawable foo \
  -title "A Sample Graph" \
  -xtitle "The X-Axis" -ytitle "The Y-Axis"
PlotPoints -drawable foo -table t \
  -xfield x -yfield y -style triangle \
  -linecolor red
```

The powerful aspect of a drawable is that it enables a user to place multiple (potentially overlapping) drawable regions onto the drawing surface. This feature can be used to implement a number of interesting graphs. For example, in Figure 3 (taken from [6]), two regions of the graph are of interest but hard to see due to their small size. Thus, one can create two additional drawables and plot closeups of the data in those regions:

These types of closeups are trivial to implement. Here is the code for one of them (the entire example, and many others, can be found on the Zplot web site):

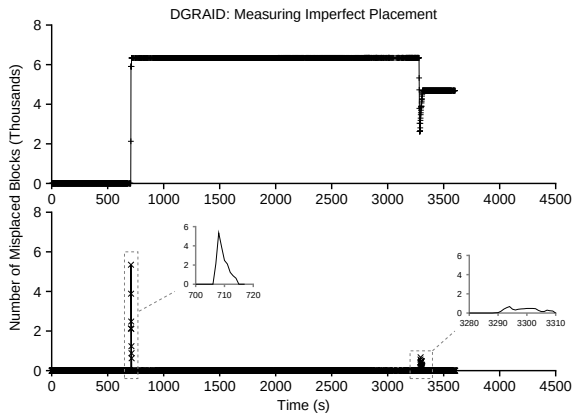

Figure 3: **Nested Plots.** *A plot from an earlier paper of ours is recreated. Two closeups are made in the lower graph, with only a few lines of Tcl code required.*

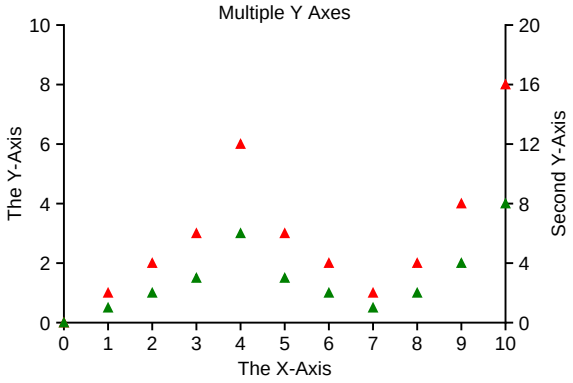

Figure 4: **Multiple Y Axes.** *The script creates two drawables, the right one with a y-range that is twice as high as the left one. The same data is plotted on both.*

```
Drawable -drawable copyc1 -coord 135,90 \
  -dimensions 40,40 -xrange 700,720 \
  -yrange 0,6000
Table -table copyc1 -columns c0,c1
TableSelect -from copy -to copyc1 \
  -where {($c0>=700) && ($c0<=720)}
AxesTicsLabels -drawable copyc1 \
  -xauto ,,10 -yauto ,,2000 \
  -linecolor gray -fontsize 6
PlotLines -drawable copyc1 -table copyc1 \
  -xfield c0 -yfield c1 -linewidth 0.25
```

This example also demonstrates a number of parameters that the `Drawable` routine can be passed. For example, a user can specify its exact position with the `coord` flag and its size with the `dimensions` parameter.

Multiple drawables can also be used to plot data with multiple y axes in a simple and straightforward manner. In this example, we plot the same data from the example above, except onto an overlapping drawable that maps the y range from 0 up to 20 (instead of 0 to 10). The code is below; the resulting graph (Figure 4) thus plots the same data twice, once in red (as relative to the left y-axis), and once in green (as relative to the right).

```
Drawable -drawable second -xrange 0,10 \
 -yrange 0,20 -width 230
AxesTicsLabels -drawable second -style y \
 -ytitle "Second Y-Axis" -labelstyle in \
 -yaxisposition 10 -yauto ,,4
PlotPoints -drawable second -table t \
 -xfield x -yfield y -style triangle \
 -linecolor green -fill t -fillcolor green
```

2.3 The Plot* Family

There are currently eight members in the `Plot*` family: `Heat`, `VerticalBars`, `HorizontalBars`, `VerticalIntervals`, `HorizontalIntervals`, `Points`, `Lines`, and `VerticalFill`. Most should be self explanatory from the name, and examples of each can be found in Figure 5.

There is also a plotting function that takes an equation instead of a table: `PlotFunction`. This routine simply takes a function to evaluate and draws the result.

2.4 Axes, Tics, and Labels

A single complex routine supports the generation of axes, tic marks, and labels for a graph. It is (not surprisingly) called `AxesTicsLabels`. It has too many arguments to describe here in any detail. However, it is often quite simple to use. For example, to specify the title, label for the x-axis, and label for the y-axis, one simple do the following:

```
AxesTicsLabels -title "Title" \
  -xtitle "X-Axis" -ytitle "Y-Axis"
```

Internal algorithms compute reasonable locations for said labels (depending on whether tic marks are used, for example). Further, when the guesses are wrong, one can use a shift argument to move the text to a more appropriate location (*e.g.*, the `-titleshift` argument can be passed the value `3,0` to bump it 3 points to the right). Many of the other options deal with customizations such as font selection, rotation, color, and so forth.

2.5 Legend

Finally, Zplot provides support in most plotting routines for the addition of a legend. A given plot routine takes an optional `-legend` flag which indicates the name to be associated with the data. The user subsequently calls the `Legend` routine, to place the legend on the screen and control its appearance.

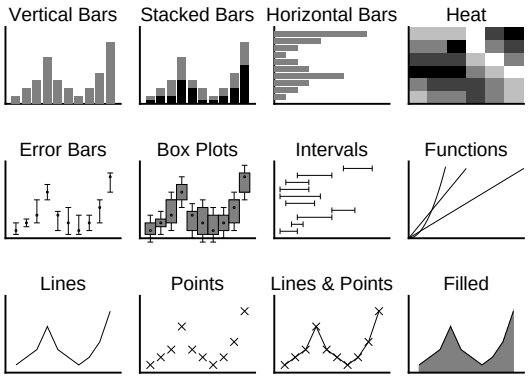

Figure 5: **Multiple Plot Types.** *This example plots a number of different plot types, as described in each title. Of course, many other variations are possible.*

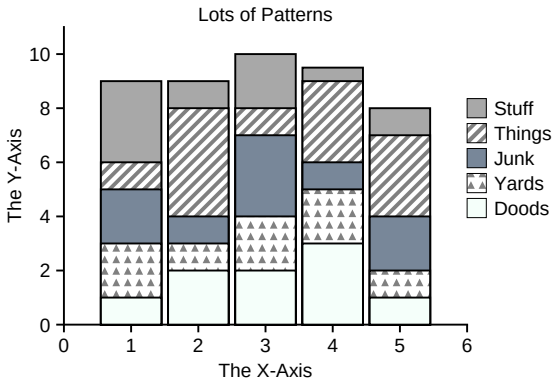

Figure 6: **Multiple Patterns.** *This example plots a number of different patterns in a set of stacked bars. As one can see, patterns such as diagonal lines and triangles can be used to fill a region. The example also includes a legend.*

3 PostScript Generation

Zplot is built on top of a number of underlying PostScript primitives, including basic lines, filled (or empty) shapes, and text. Each of these routines is used by the plotting routines and other entities that wish to create graphical or textual elements upon on the drawing surface. We now describe the primitives in turn.

```
PsLine -coord <x1,y1:x2,y2:...:xN,yN>
   -linecolor <color>
   -linewidth <width in pts>
   -linecap <0, 1, or 2>
   -linejoin <0, 1, or 2>
   -linedash <dash pattern>
   -closepath <true or false>
```

The `PsLine` primitive is passed a set of coordinates, some basic information about the line, and then produces a line that connects the coordinates in the resulting PostScript. All PostScript primitives take coordinates in PostScript "ems", each of which is 1/72nd of an inch. The `PsLine` primitive also takes additional arguments that allow the addition of an arrow to the end of the line; we omit these parameters for the sake of space.

```
PsBox -coord x1,y2:x2,y2
PsCircle -coord x,y -radius r
PsPolygon -coord x1,y1:...:xN,yN
  -linecolor
  -linedash
  -linecap
  -fill <true or false>
  -fillcolor <color of each element>
  -fillstyle <style>
  -fillsize <size of element in pattern>
  -fillskip <amount to skip between ...>
  -fillshift <+x,+y>
  -bgcolor <color behind pattern>
```

Each of these shape routines take a variety of arguments that describe their coordinates, and then all take three different sets of arguments that characterize the line around the shape (`-line*`), the fill of the shape (`-fill*`), and the background color behind the shape (-bgcolor). The line descriptors match those of `PsLine` above, and the background color is straightforward. Most interesting, then, is the variety and flexibility provided by the pattern descriptions.

The `-fill*` parameters allows users to specify a fill pattern for a region. The most important parameter is `-fillstyle`, which determines how the region is filled. Current styles that are supported include `solid, hline, vline, dline1, dline2, circle, square, triangle, utriangle`; more are added occasionally (when the author needs them). Each pattern takes two arguments to determine its contents: `-fillsize` and `-fillskip`. Within a given pattern, `-fillsize` determines the size of each element in the pattern, and `-fillskip` the space between each element. Figure 6 is a bar graph that demonstrates the use of some of these patterns.

```
PsText -coord <x,y>
  -text <the text to write on canvas>
  -font <font choice>
  -color <color>
  -rotate <angle of rotation>
  -anchor <how to anchor the text>
  -bgcolor <background color behind text>
  -bgborder <size of border around text>
```

The last primitive we describe is `PsText`, which draws text onto the screen. Most of its parameters are straightforward. However, the most crucial argument to understand is the `anchor`. This parameter describes how the

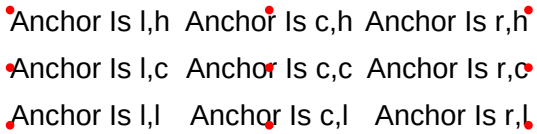

Figure 7: **Text Anchors.** *This example shows how to specify text anchors.*

text should be anchored relative to the coordinate that was passed to the routine. The parameter takes the form `xanchor,yanchor`, where xanchor specifies the anchoring of the text in the x direction (either `l` for left, `c` for center, or `r` for right), and yanchor the anchoring in the y direction (`l` for low, `c` for center, and `h` for high). Figure 7 shows the different possible anchors (the coordinates passed to the text drawing routine are highlighted with a red circle).

4 Commenting on Tcl

We now comment on a few aspects of Tcl that arose during the implementation of Zplot. We begin with performance issues, comment on namespaces and packages, and finally discuss error checking.

4.1 Performance

As floating point specialist William Kahan famously said, "The fast drives out the slow, even if the fast is wrong." Tcl is slow. Thus, Zplot is slow. If one tries to plots graphs with thousands of data points, one will have to wait, even on a modern processor. To show how slow, I present a rudimentary performance study of Zplot performance.

In the experiment, I simply timed how long it takes to produce a plot given an input file with 100,000 data points. The experiment was run upon a MacBook Pro laptop with 2.16 GHz Intel Core 2 Duo processors, 1 GB of RAM, and running Mac OS X 10.4.9. Five trials were run, and the input file fit comfortably into main memory (thus, no substantial I/O activity occurs during the experiment).

The average time to run Zplot over this large data file was 45.93 seconds (with very little variation). In comparison with other tools written in C, Zplot performance is many orders of magnitude slower (*e.g.*, plotting the same input file with gnuplot is nearly instantaneous). It is true that Zplot was not written with optimized performance in mind, but it was not written to be horrifically slow, either. It is simply the case that building clean Tcl programs with many nested subroutine calls leads to poor performance.

Ironically, John Ousterhout's paper [5] points out many reasons that operating system performance does not scale with processor performance; analogous arguments can be made about Tcl. Although processors have improved greatly in the past 10 years, Tcl remains slow in both a relative and absolute sense. It is this author's opinion that this performance flaw is one major reason Tcl has not become more broadly accepted.

4.2 Namespaces and Packages

Tcl namespaces are a simple and powerful feature; as a long-time Tcl user, they have been a welcome addition. Somehow, I do not find myself using Tcl packages; instead, I just create a single large Tcl file from the various source files of Zplot, and `source` said file to use Zplot. Primitive? Certainly. And yet somehow I prefer it to the current package creation system.

4.3 Error Checking

I found myself cursing the lack of assistance for error checking in Tcl. For example, when a user calls a routine and accidentally passes text instead of a numeric value to a particular routine, if one is not careful, some kind of Tcl error message will get printed and the program aborted – not very user-friendly.

To cope with this problem, I wrote a generic argument parsing package that performed type checking and other type-specific checks on a per-argument basis. Internally, most user-callable routines begin with a declaration as follows:

```
proc Table {args} {
  set default {
   {"table"   "default" \
      "isString 1" "name to call table"}
   {"file"    ""        \
      "isFile 1"   "file to read from"}
   {"separator" ""      "isString 1" \
      "if empty, whitespace; \
       otherwise, whatever is specified"}
  }
  ArgsProcess Table default args use \
  "Create a table. If '-file' is specified, \
   load the table from a file. Otherwise, \
   '-columns' must be specified and give a \
   comma-separated list of columns in the \
   table (e.g., '-columns x,y,mean')."
  ...
```

For each argument, a routine is specified that is used to perform whatever checks are relevant. For example, for the `-table` parameter above, the routine `isString` is called to ensure that the table name is a string (a primitive perform of dynamic type-checking). Defaults are also specified in case the user does not specify a given argument (*e.g.*, `-table` will default to the `default` table). When a problem occurs, an error message prints out each parameter, its default value, the info string per parameter (*e.g.*, `name to call the table`), and the overall description of the function as specified in the call to `ArgsProcess`. As mentioned above, one can call most routines with a bad flag to obtain said information.

5 Related Work

Much of the frustration I spoke of earlier was with a tool known as gnuplot [7]. Gnuplot provides excellent support for simple line graphs and scatter plots, as well as numerous other graph types. However, its lack of reasonable support for bar charts was one of the main driving forces behind Zplot. However, I should note that the PostScript produced by gnuplot was clear and easy to read, sparking my interest in that language, and thus (indirectly) making Zplot possible. Great PostScript resources, for those who are interested, are the blue book, red book, and (to some extent), the green book [1, 3, 2]; all are available online.

As I demonstrated Zplot to others, many people referred me to Ploticus [4], which is a more powerful and complete tool than gnuplot and is capable of producing a large variety of interesting graph types. Many of the features found in Zplot are also found in ploticus (e.g., a ploticus "area" is akin to a Zplot Drawable), and I often found myself downloading examples from the Ploticus web page to see if Zplot could easily do what Ploticus already does. Indeed, at one point I even considered dropping Zplot development and simply adding a few features to Ploticus that I found lacking (*e.g.*, bar graphs with a variety of pretty patterns). However, one look at the Ploticus source code convinced me that I might be on the right path (or, at least, a different path). Ploticus is comprised of over 60,000 lines of C code. Zplot, in contrast, is less than 5,000 lines of Tcl; although not always the prettiest code, certainly quite a bit simpler. This comparison is certainly a bit unfair, as Zplot is not as powerful as Ploticus, but I feel quite positive that it will never be nearly as large or complex, a testimony to the power of a higher-level language such as Tcl.

6 Future Work

Zplot is incomplete in a number of ways. For example, although the PostScript it generates is simple, it is often inefficient (*i.e.*, the resultant PostScript is larger than it need be). Some simple optimizations would noticeably reduce the size of the resultant PostScript files.

Error reporting has improved throughout the course of Zplot's development, but could always be better. The development of a more powerful argument processing package (as described above) helped a great deal, but there are still some cases where a user could trigger an internal assertion to fail and thus will see a stack trace telling them where something went wrong. Better error reporting remains something I plan to look into.

Finally, there are a host of features which would be useful. Better support for time and date formats would be of great benefit. More line styles, point styles, and fill patterns are always helpful. A facility to automate graph generation (much like the "prefabs" offered by ploticus) would probably be well-received.

7 Conclusions

In this paper, I have introduced Zplot, a pure Tcl package for drawing PostScript figures. Zplot provides a number of powerful but simple tools for making beautiful two-dimensional plots. In the course of building Zplot, I was again surprised by how slow Tcl is; however, its simplicity and power make programming in Tcl something unusual (to me) among its counterparts: fun.

If you are interested in Zplot, please visit: `www.zplot.org`.

Acknowledgments

The author thanks his colleagues at the University of Michigan for all of their support during the sabbatical year which made Zplot possible. The author also thanks his wife for the numerous discussions she was forced to have about Zplot, which she did gladly and gracefully, whether she wanted to or not. Finally, the author thanks his two daughters, Anna and Maddy, for looking at some of the resulting graphs and "oohing" and "ahhing" at the appropriate times.

References

[1] Adobe Systems Inc. PostScript Language Tutorial and Cookbook. www-cdf.fnal.gov/offline/PostScript/BLUEBOOK.PDF, 1985.

[2] Adobe Systems Inc. PostScript Language Program Design. www-cdf.fnal.gov/offline/PostScript/GREENBK.PDF, 1988.

[3] Adobe Systems Inc. PostScript Language Reference Manual. www-cdf.fnal.gov/offline/PostScript/PLRM2.pdf, 1990.

[4] Stephen C. Grubb. Ploticus. ploticus.sourceforge.net, 2007.

[5] John K. Ousterhout. Why Aren't Operating Systems Getting Faster as Fast as Hardware? In *Proceedings of the 1990 USENIX Summer Technical Conference*, Anaheim, CA, June 1990.

[6] Muthian Sivathanu, Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Improving Storage System Availability with D-GRAID. In *Proceedings of the 3rd USENIX Symposium on File and Storage Technologies (FAST '04)*, pages 15–30, San Francisco, California, April 2004.

[7] Thomas Williams, Colin Kelley, Russell Lang, Dave Kotz, John Campbell, Gershon Elber, and Alexander Woo. Gnuplot. www.gnuplot.info, 2007.