PLoT Scheme

Alexander Friedman and Jamie Raymond
College of Computer Science
Northeastern University
Boston, MA 02115
USA
{cozmic,raymond}@ccs.neu.edu

ABSTRACT

We present PLTplot, a plotting package for PLT Scheme. PLTplot provides a basic interface for producing common types of plots such as line and vector field plots from Scheme functions and data, an advanced interface for producing customized plot types, and support for standard curve fitting. It incorporates renderer constructors, transformers from data to its graphical representation, as values. Plots are also values. PLTplot is built as an extension on top of the third-party PLplot C library using PLT Scheme's C foreign function interface. This paper presents the core PLTplot API, examples of its use in creating basic and customized plots and fitting curves, and a discussion of its implementation.

1 INTRODUCTION

This paper describes PLTplot a plotting extension for PLT Scheme [6] based on the PLplot [5] C library. PLTplot is provided as a set of modules that provide language constructs and data types for producing plots of functions and data. The basic interface provides constructors for rendering data and functions in common forms such as points and lines. For advanced users, PLTplot provides an interface for building custom renderer constructors on top of basic drawing primitives to obtain almost any kind of desired plot.

Our motivation for producing PLTplot was to be able to do plotting from within our favorite programming language, Scheme, instead of our usual method of only using Scheme to work with the data and then calling an external program, such as Gnuplot [10], to actually produce the plots. This mechanism was tedious, especially when we only wanted a quick and dirty plot of a Scheme function or data contained in a Scheme list or vector.

To develop the package as quickly as possible, instead of creating a plot library on top of PLT Scheme's graphical toolkit, MrEd, which would have meant a lot of engineering work, we decided to reuse the heavy lifting already done by the visualization experts. We looked at existing plotting packages and libraries written in C with appropriate free source licensing on which we could build an extension for PLT Scheme using its C foreign function interface (FFI).

Initially we considered using Gnuplot, but it builds only as a monolithic executable and not as a library. Modifying it was too daunting as it has an extremely baroque codebase. We looked elsewhere and found an LGPLed plotting library called PLplot [5] that has been used as an extension by several other programming languages. PLplot is currently being developed as a Sourceforge project; it provides many low-level primitives for creating 2D and 3D graphs of all sorts. With PLplot's primitives wrapped as Scheme functions as a foundation, we created a high-level API for plotting and included some additional utilities such as curve fitting.

2 PLTPLOT IN ACTION

PLTplot supports plotting data and functions in many common forms such as points, lines, or vector fields. It also supports the creation of custom renderers that can be used to visualize the data in new ways. Data can be fitted to curves and the resulting functions plotted along with the original data. We illustrate these ideas with an example from physics.

2.1 Simple Plots

Figure 1 shows a plot of data [9] similar to that collected by Henry Cavendish in 1799 during his famous experiment to weigh the earth. The X axis represents time, while the Y axis represents the angle of rotation of Cavendish's torsional pendulum. The data is defined as a list of Scheme vectors, each one containing a value for time, angle of rotation, and error.

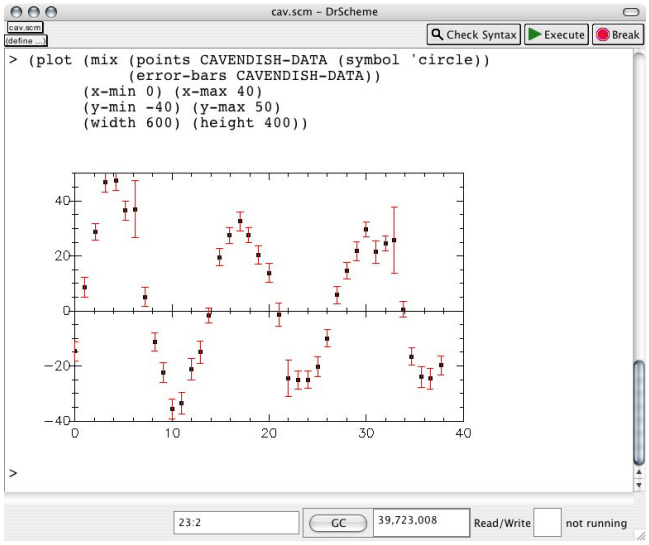
```
(define CAVENDISH-DATA
  (list (vector 0.0 -14.7 3.6)
        (vector 1.0 8.6 3.6)
        ...                      ;; omitted points
        (vector 37.7 -19.8 3.5)))
```

Before the data can be plotted, the plot module must be loaded into the environment using the following code:

```
(require (lib "plot.ss" "plot"))
```

As illustrated in Figure 1, the plot is generated from code entered at the REPL of DrScheme, PLT Scheme's programming environment. `points` and `error-bars` are constructors for values called Plot-items that represent how the data should be rendered. The call to `mix` takes the two Plot-items and composes them into a single Plot-item that `plot` can display. The code ends with a sequence of associations that instruct `plot` about the range of the X and Y axes and the dimensions of the overall plot.

Figure 1 Points and Error Bars



2.2 Curve Fitting

From the plot of angles versus time, Cavendish could sketch a curve and model it mathematically to get some parameters that he could use to compute the weight of the earth. We use PLTplot's built-in curve fitter to do this for us.

To generate a mathematically precise fitted curve, one needs to have an idea about the general form of the function represented by the data and should provide this to the curve fitter. In Cavendish's experiment, the function comes from the representation of the behavior of a torsional pendulum. The oscillation of the pendulum is modeled as an exponentially decaying sinusoidal curve given by the following equation: $f(s) = \theta_0 + ae^{\frac{-s}{\tau}} \sin(\frac{T}{2\pi s} + \phi)$. The goal is to come up with values for the constant parameters in the function so that the phenomena can be precisely modeled. In this case, the fit can give a value for T, which is used to determine the spring constant of the torsional fiber. This constant can be used to compute the universal gravitational constant – the piece Cavendish needed to compute the weight of the earth.

PLTplot fits curves to data using a public-domain implementation of the the standard Non-Linear Least Squares Fit algorithm. The user provides the fitter with the function to fit the data to, hints and names for the constant values, and the data itself. It will then produce a `fit-result` structure which contains, among other things, values for the contants and the fitted function with the computed parameters.

Figure 2 shows the code for generating the fit and producing a new plot that includes the fitted curve. To get the value of the parameter T, we select the values of the final parameters from `result` with a call to `fit-result-final-params`, shown in the code, and then inspect its output for the value.

Figure 2 Mixed Plot with Fitted Curve

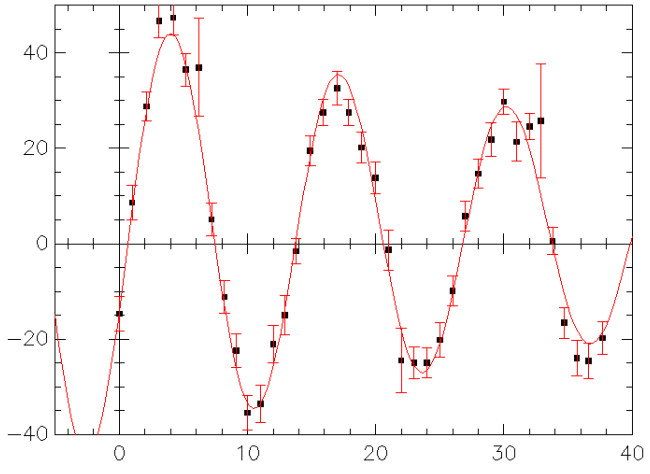
```
(require (lib "plot.ss" "plot"))

(define theta
  (lambda (s a tau phi T theta0)
    (+ theta0
       (* a
          (exp (/ s tau -1))
          (sin (+ phi (/ (* 2 pi s) T)))))))

(define result
  (fit
   theta
   ((a 40) (tau 15) (phi -.5) (T 15) (theta0 10))
   CAVENDISH-DATA))

(fit-result-final-parms result)

(plot (mix
        (points CAVENDISH-DATA)
        (error-bars CAVENDISH-DATA)
        (line (fit-result-function result)))
      (x-min -5) (x-max 40)
      (y-min -40) (y-max 50))
```



2.3 Complex Plots

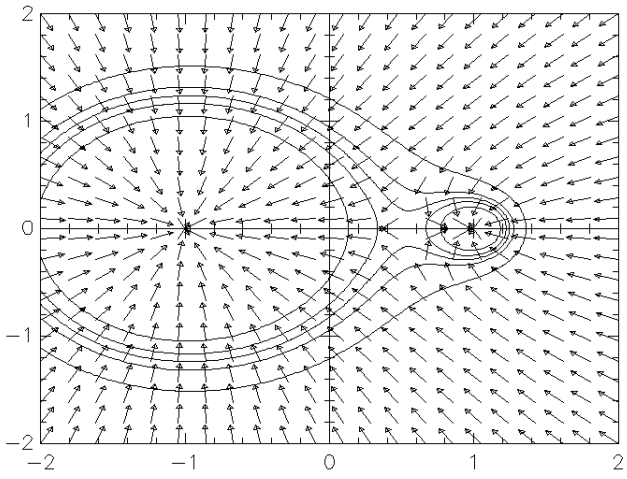
Besides operations to produce simple point and line plots, PLTplot also supports more complex plot operations. In this next example we have an equation which represents the gravitational potential of two bodies having unequal masses located near each other. We plot this equation as both a set of contours and as a vector field with several non-default options to better visualize the results. For the contour part of the plot, we manually set the contour levels and the number of times the function is sampled. For the vector part, we numerically compute the gradient of the function and reduce the number of vectors displayed to 22 and change the style of the vectors to normalized. The code and resulting plot are shown in Figure 3.

Figure 3 Contour and Vector Field

```
(require (lib "plot.ss" "plot"))

(define gravitational-potential
  (lambda (x y)
    (-
     (/ -1
        (sqrt (+ (sqr (add1 x)) (sqr y))))
     (/ 1/10
        (sqrt (+ (sqr (sub1 x)) (sqr y)))))))

(plot
 (mix (contour
       gravitational-potential
       (levels '(-0.7 -0.8 -0.85 -0.9 -1))
       (samples 100))
      (field
       (gradient
         (lambda (x y)
           (* -1 (gravitational-potential x y))))
       (samples 22) (style 'normalized)))
 (x-min -2) (x-max 2)
 (y-min -2) (y-max 2))
```



3 CORE PLTPLOT

PLTplot is meant to be easy to use and extensible. The functionality is naturally split into two levels: a basic level, which provides a set of useful constructors that allow creation of common types of plots, and an advanced level, which allows the creation of custom renderer constructors. The API for core PLTplot is shown in Figure 4.

PLTplot, in addition to being a library for PLT Scheme, is a little language for plotting. The idea is to keep the process of plotting a function or data set as simple as possible with as little decoration as necessary. This is a cognitive simplification for the casual PLTplot user, if not also a syntactic one. The special form `plot`, for instance, takes a Plot-item (constructed to display the data or function in a particular way, such as a line or only as points) followed by a possibly empty sequence of attribute-value associations. If `plot` were a function, these associations would have to be specially constructed as Scheme values, for examples as lists, which would necessitate decoration that is irrelevant to specifying the specific features of the generated plot. Other forms are provided for similar reasons.

3.1 Basic Plotting

The fundamental datatype in PLTplot is the Plot-item. A Plot-item is a transformer that acts on a view to produce a visual representation of the data and options that the Plot-item was constructed with. An interesting and useful feature of the constructed values that Plot-items produce is that they are functionally composable. This is practically used to produce multiple renderings of the same data or different data on the resulting view of the plot. Plot-items are displayed using the `plot` special form. `plot` takes a Plot-item and some optional parameters for how the data should be viewed and produces an object of the `2d-view%` class which DrScheme displays.

Plot-items are constructed according to the definitions shown in Figure 4. Consider the `line` constructor. It consumes a function of one argument and some options and produces a transformer that knows how to draw the line that the function represents. Its options are a sequence of keyword-value associations. Some possible `line-option`s include `(samples number)` and `(width number)`, specifying the number of times the function is sampled and the width of the line, respectively. Each of the other constructors have similar sets of options, although the options are not necessarily shared between them. For example, the `samples` option for a line has no meaning for constructors that handle discrete data.

The other Plot-item constructors grouped with `line` in the definition of Plot-item are used for other types of plots. `points` is a constructor for data representing points. `error-bars` is a constructor for data representing points associated with error values. `shade` is a constructor for a 3D function in which the height at a particular point would be displayed with a particular color. `contour` is likewise a constructor for a 3D function that produces contour lines at the default or user-specified levels. And `field` is a constructor for a function that represents a vector field.

The `mix` constructor generates a new Plot-item from two or more existing Plot-items. It is used to combine multiple

Figure 4 Core PLTplot API

```
Data Definitions

A Plot-item is one of
 (line 2dfunction line-option*)
 (points (list-of (vector number number))
         point-option*)
 (error-bars
    (list-of (vector number number number))
    error-bar-option*)
 (shade 3dfunction shade-option*)
 (contour 3dfunction countour-option*)
 (field R2->R2function field-option*)

 (mix Plot-item Plot-item+)
 (custom (2d-view% -> void))

*-option is: (symbol TST)
  where TST is any Scheme type

A 2dfunction is (number -> number)

A 3dfunction is (number number -> number)

A R2->R2function is one of
  ((vector number number) ->
   (vector number number))
  (gradient 3d-function)

fit-result is a structure:
  (make-fit-result ... (list-of number) ...
                   (number* -> number))

2d-view% is the class of the displayed plot.

Forms

(plot Plot-item (symbol number)*)

(fit (number* -> number)
     ((symbol number)*)
     (list-of
      (vector number [number] number number)))

(define-plot-type name data-name view-name
                       ((option value)*)
                  body)

Procedures

(fit-result-final-params fit-result) ->
  (list-of number)
(fit-result-function fit-result) ->
  (number* -> number)
... additional fit-result selectors elided ...
```

items into a single one for plotting. For example in Figure 2 `mix` was used to combine a plot of points, error bars, and the best-fit curve for the same set of data.

The final Plot-item constructor, `custom`, gives the user the ability to draw plots on the view that are not possible with the other provided constructors. Programming at this level gives the user direct control over the graphics drawn on the plot object, constructed with the `2d-view%` class.

3.2 Custom Plotting

The `2d-view%` class provides access to drawing primitives. It includes many methods for doing things such as drawing lines of particular colors, filling polygons with particular patterns, and more complex operations such as rendering data as contours.

Suppose that we wanted to plot some data as a bar chart. There is no bar chart constructor, but since we can draw directly on the plot via the `custom` constructor we can create a bar with minimal effort.

First we develop a procedure that draws a single bar on a `2d-view%` object.

```
; draw an individual bar
(define (draw-bar x-position width height view)
  (let ((x1 (- x-position (/ width 2)))
        (x2 (+ x-position (/ width 2))))
   (send view fill
         `(,x1 ,x1 ,x2 ,x2)
         `(0 ,height ,height 0))))
```

Then we develop another procedure that when applied to an object of type `2d-view%` would draw all of the data, represented as a list of two-element lists, using `draw-bar` on the view.

```
; size of each bar
(define BAR-WIDTH .75)

; draw a bar chart on the view
(define (my-bar-chart 2dview)
  (send 2dview set-line-color 'red)
  (for-each
   (lambda (bar)
     (draw-bar (car bar) BAR-WIDTH
               (cadr bar) 2dview))
   '((1 5) (2 3) (3 5) (4 9) (5 8)))) ; the data
```

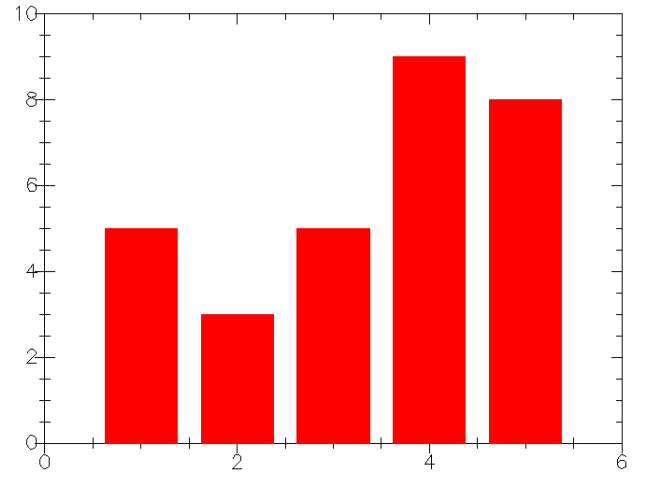
We then create the plot using the `plot` form, wrapping `my-bar-chart` with the `custom` constructor.

```
(plot (custom my-bar-chart)
      (x-min 0) (y-min 0) (x-max 6) (y-max 10))
```

The results are shown in Figure 5. The output is plain but useful. We could enhance the appearance of this chart using other provided primitives. For example, the bars could be drawn with borders, the axes given labels, etc.

While we now get the desired bar chart, we have to change the data within the `my-bar-chart` procedure each time we want a new chart. We would like to abstract over the code for `my-bar-chart` to create a generic constructor similar to the built-in ones. To manage this we use a provided special form, `define-plot-type`, which is provided in the module

Figure 5 Custom Bar Chart



`plot-extend.ss`. It takes a name for the new plot type, a name for the data, a name for the view, an optional list of fields to extract from the view, and a set of options with default values. It produces a Plot-item constructor. We can now generalize the above function as follows:

```
(require (lib "plot-extend.ss" "plot"))

(define-plot-type bar-chart
  data 2dview [(color 'red) (bar-width .75)]
    (begin
      (send 2dview set-line-color color)
      (for-each
       (lambda (bar) (draw-bar (car bar) bar-width
                               (cadr bar) 2dview))
       data)))
```

Our original data definition for Plot-item can now be augmented with the following:

```
(bar-chart (list-of (list number number))
           [(color symbol) (bar-width number)])
```

Plotting the above data with blue bars would look like:

```
(plot
  (bar-chart
    '((1 5) (2 3) (3 5) (4 9) (5 8))
    (color 'blue))
  (x-min 0) (y-min 0) (x-max 6) (y-max 10))
```

3.3 Curve Fitting API

Any scientific plotting package would be lacking if it did not include a curve fitter. PLTplot provides one through the use of the `fit` form. `fit` must be applied to a function (a representation of the model), the names and guesses for the parameters of the model, and the data itself. The guesses for the parameters are hints to the curve fitting algorithm to help it to converge. For many simple model functions the guesses can all be set to 1.

The data is a list of vectors. Each vector represents a data point. The first one or, optionally, two elements of a data vector are the values for the independent variable(s). The last two elements are the value for the dependent variable and the weight of its error. If the errors are all the same, they can be left as the default value 1.

The result of fitting a function is a `fit-result` structure. The structure contains the final values for the parameters and the fitted function. These are accessed with the selectors `fit-result-final-params` and `fit-result-function`, respectively. The structure contains other useful values as well that elided here for space reasons but are fully described in the PLTplot documentation.

4 IMPLEMENTATION

PLTplot is built on top of a PLT Scheme extension that uses the PLplot C library for its plotting and math primitives. To create the interface to PLplot, we used PLT Scheme's C FFI to build a module of Scheme procedures that map to low-level C functions. In general, the mapping was straight forward – most types map directly, and Scheme lists are easily turned into C arrays.

Displayed plots are objects created from the `2d-view%` class. This class is derived from PLT Scheme's `image-snip%` class which gives us, essentially for free, plots as first-class values. In addition `2d-view%` acts as a wrapper around the low level module. It provides some error checking and enforces some implied invariants in the C library.

It was important that PLTplot work on the major platforms that PLT Scheme supports: Windows, Unix, and Mac OS X. To achieve this we used a customized build process for the underlying PLplot library that was simplified by using `mzc` – the PLT Scheme C compiler – which generated shared libraries for each platform.

5 RELATED WORK

There is a tradition in the Scheme community of embedding languages in Scheme for drawing and graphics. Brian Beckman makes the case that Scheme is a good choice as a core language and presents an embedding for doing interactive graphics [2]. Jean-François Rotgé embedded a language for doing 3D algebraic geometry modeling [8]. We know of no other published work describing embedding a plotting language in Scheme.

One of the most widely used packages for generating plots for scientific publication is Gnuplot, which takes primitives for plotting and merges them with an ad-hoc programming language. As typically happens in these cases the language grows from something simple, say for only manipulating columns of data, to being a full fledged programming language. Gnuplot is certainly an instance of Greenpun's Tenth Rule of Programming: "Any sufficiently complicated C or Fortran program contains an ad-hoc, informally-specified bug-ridden slow implementation of half of Common Lisp." [3]. What you would rather have is the marriage of a well-documented, non-buggy implementation of an expressive programming language with full-fledged plotting capabilities.

Some popular language implementations, like Guile and Python, provide extensions for Gnuplot. This is a step forward because now one can use his or her favorite program-

ming language for manipulating data and setting some plot options before shipping it all over to Gnuplot to be plotted. However, the interface for these systems relies on values in their programming languages being translated into Gnuplot-ready input and shipped as strings to an out-of-process Gnuplot instance. For numerical data this works reasonably well, but if functions are to be plotted, they must be written directly in Gnuplot syntax as strings, requiring the user to learn another language, or be parsed and transformed into Gnuplot syntax, requiring considerable development effort.

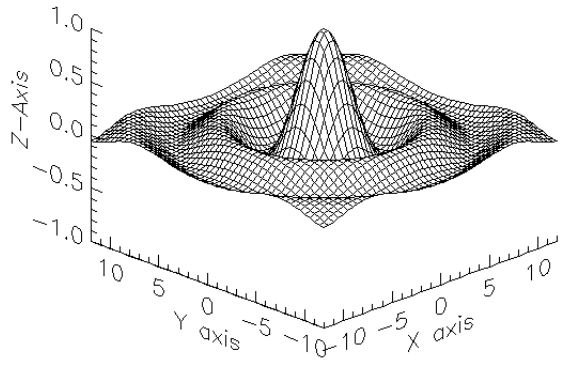
The idea behind our integration was to join the well-specified and well-documented PLT Scheme with a relatively low-level library for drawing scientific plots, PLplot. We then could handle both data and options in the implementation in a rich way rather than as plain strings. We then provided a higher-level API on top of that. Other platforms that have integrated PLplot only provide the low-level interface. These include C, C++, Fortran-77, Tcl/TK, and Java among others.

Ocaml has a plotting package called Ocamlplot [1], which was built as an extension to libplot, part of GNU plotutils [4]. libplot renders 2-D vector graphics in a variety of formats and can be used to create scientific plots but only with much effort by the developer. Ocamlplot does not provide a higher-level API like PLTplot does. For example, there is no abstraction for line plots or vector plots. Instead the user is required to build them from scratch and provide his own abstractions using the lowest-level primitives. There is also no notion of first class plots as plots are output to files in specific graphic formats.

6 CONCLUSION

This paper presented core PLTplot, which is a subset of what PLTplot provides. In addition to the core 2D API illustrated in this paper, PLTplot also provides an analogous API for generating 3D plots, an example of which is seen in Figure 6.

Figure 6 $\frac{\sin(\sqrt{x^2+y^2})}{\sqrt{x^2+y^2}}$



PLTplot is still new and many additions are planned for the future. With the publication of this paper, the first release of PLTplot will be made available through the PLT Scheme Libraries and Extensions website [7]. Currently plots are only output as `2dview%` objects. One addition we hope to make soon is the ability to save plots in different formats including Postscript. We also plan on developing a separate plotting environment which will have its own interface for easily generating, saving, and printing plots.

7 ACKNOWLEDGMENTS

The authors would like thank Matthias Felleisen for his advice during the design of PLTplot and subsequent comments on this paper. They also thank the anonymous reviewers for their helpful comments.

REFERENCES

[1] Andrieu, O. Ocamlplot [online]. Available from World Wide Web: `http://ocamlplot.sourceforge.net`.

[2] Beckman, B. A scheme for little languages in interactive graphics. `Software-Practice and Experience 21` (Feb. 1991), 187–207.

[3] Greenspun, P. Greenspun's Tenth Rule of Programming [online]. Available from World Wide Web: `http://philip.greenspun.com/research/`.

[4] Maier, R., and Tufillaro, N. Gnu plotutils [online]. Available from World Wide Web: `http://www.gnu.org/software/plotutils`.

[5] PLplot core development team. PLplot [online]. Available from World Wide Web: `http://plplot.sourceforge.net`.

[6] PLT. PLT Scheme [online]. Available from World Wide Web: `http://www.plt-scheme.org`.

[7] PLT. Plt scheme libraries and extensions [online]. Available from World Wide Web: `http://www.cs.utah.edu/plt/develop/`.

[8] Rotgé, J.-F. SGDL-Scheme: a high level algorithmic language for projective solid modeling programming. In `Proceedings of the Scheme and Functional Programming 2000 Workshop` (Montreal, Canada, Sept. 2000), pp. 31–34.

[9] Vrable, M. Cavendish data [online]. Available from World Wide Web: `http://www.cs.hmc.edu/~vrable/gnuplot/using-gnuplot.html`.

[10] Williams, T., and Hecking, L. Gnuplot [online]. Available from World Wide Web: `http://www.gnuplot.info`.