PLoT Manual

PLT (scheme@plt-scheme.org)

207
Released May 2004

Copyright notice

Contents

1. Quick Start

1.1 Overview

PLoT (aka PLTplot) provides a basic interface for producing common types of plots such as line and vector field plots as well as an advanced interface for producing customized plot types. Additionally, plots and plot-items are first-class values and can be generated in and passed to other programs.

1.2 Basic Plotting

After loading the correct module using (**require** (*lib* "plot.ss" "plot")) try (*plot* (*line* (**lambda** (*x*) *x*)))

Any other function with the contract *number* -> *number* can be plotted using the same form. To plot multiple items, use the functions *mix* and *mix∗* to combine the items to be plotted

```
(plot (mix (line (lambda (x) (sin x)))
           (line (lambda (x) (cos x)))))
```

The display area and appearance of the plot can be changed by adding parenthesized argument/value pairs after the first argument.

```
(plot (line (lambda (x) (sin x)))
      (x-min −1)
      (x-max 1)
      (title "Sin(x)"))
```

The appearance of each individual plot item can be altered by adding parameter-value pairs after the data.

```
(plot (line (lambda (x) x)
            (color 'green)
            (width 3)))
```

Besides plotting lines from functions in 2d, the plotter can also render a variety of other datums in several ways:

- Discreet data, such as

  ```
  (define data
   (list (vector 1 1 2)
         (vector 2 2 2)))
  ```

  can be interpreted in several ways:

  - As points: (*plot* (*points data*))
  - As Error Data: (*plot* (*error-bars data*))

- A function of two variables, such as

  (**define** *3dfun* (**lambda** (*x y*) (* (*sin x*) (*sin y*))))

  can be plotted on a 2d graph
  - Using contours to represent height (z)

    (*plot* (*contour 3dfun*))
  - Using color shading

    (*plot* (*shade 3dfun*))
  - Using a gradient field

    (*plot* (**field** (*gradient 3dfun*)))

  or in a 3d box
  - Displaying only the top of the surface

    (*plot3d* (*surface 3dfun*))

1.3 Curve Fitting

The scheme-plot library uses a Non-Linear Least Squares fit algorithm to fit parametrized functions to given data.

To fit a particular function to a curve:

1. Set up the independent and dependent variable data. The first item in each vector is the independent var, the second is the result. The last item must is the weight of the error — we can leave it as 1 since all the items weigh the same.

   ```
   (define data '(#(0 3 1)
                  #(1 5 1)
                  #(2 7 1)
                  #(3 9 1)
                  #(4 11 1)))
   ```

2. Set up the function to be fitted using fit. This particular function looks like a line. The independent variables must come before the parameters.

   ```
   (define fit-fun
     (lambda (x m b) (+ b (* m x))))
   ```

3. If possible, come up with some guesses for the values of the parameters. The guesses can be left as one, but each parameter must be named.

4. Do the fit – the details of the function are described in the Curve Fitting section

   ```
   (define fit-result
     (fit fit-fun
          ((m 1) (b 1))
          data))
   ```

5. View the resulting parameters

   (*fit-result-final-params fit-result*) ; will produce ((m 2) (b 3))

6. For some visual feedback of the fit result, plot the function with the new parameters. For convenience, the structure that is returned by the fit command has already created the function.

```
(plot (mix (points data)
           (line (fit-result-function fit-result)))
      (y-max 15))
```

A more realistic example can be found in **demos/fit-demo-2.ss**.

1.4 Creating Custom Plots

Defining custom plots is simple: a Plot-item (that is passed to plot or mix) is just a function that acts on a view. Both the 2d and 3d view snip have several drawing functions defined that the plot-item can call in any order. The full details of the view interface can be found in the **plot-extend.ss** section.

For example, if we wanted to create a constructor that creates Plot-items that draw dashed-lines given a number-number function we could do the following: Load the required modules

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

Set up the constructor

```
(define-plot-type dashed-line
  fun 2dview (x-min x-max) ((samples 100) (segments 20) (color 'red) (width 1))
  (let* ((dash-size (/ (- x-max x-min) segments))
         (x-lists (build-list
                   (/ segments 2)
                   (lambda (index)
                     (x-values
                      (/ samples segments)
                      (+ x-min (* 2 index dash-size))
                      (+ x-min (* (add1 ( * 2 index)) dash-size)))))))
    (send* 2dview
           (set-line-color color)
           (set-line-width width))
    (for-each
     (lambda (dash)
       (send 2dview plot-line
             (map (lambda (x) (vector x (fun x))) dash)))
     x-lists)))
```

Plot a test case

```
(plot (dashed-line (lambda (x) x) (color 'blue)))
```

2. Module: plot.ss

The **plot.ss** module provides the ability to make basic plots, fit curves to data, and some useful miscellaneous functions.

2.1 Plotting

The *plot* and *plot3d* forms generate plots that can be viewed in the DrScheme Interactions window. The functions and data definitions for this module are as follows:

Forms:
(*plot 2d-plot-item 2d-plot-option∗*) -> *VIEW*
(*plot3d 3d-plot-item 3d-plot-option∗*) -> *VIEW*

2d-plot-option is one of:
(*x-min number*)
(*x-max number*)
(*y-min number*)
(*y-max number*)
(*x-label string*)
(*y-label string*)
(*title string*)

3d-plot-option is one of:
2d-plot-option
(*z-label*)
(*z-min number*)
(*z-max number*)
(*alt number*) ; altitude angle, in degrees
(*az number*) ; azimuthal angle, in degrees

The 2d and 3d plot-options modify the view in which the graph is drawn. The 3d-plot-options alt and az set the viewing altitude (in degrees) and azimuth (also in degrees) respectively. The rest of the options should be self-explanatory.

Data Definitions:

2d-plot-item is one of:
(*points* (*list-of* (*vector number number*)) *point-options∗*)
(*line* [(*number -> number*) | (*number -> (vector number number*))] *line-options∗*)
(*error-bars* (*list-of* (*vector number number number*)) *error-bar-options∗*)
(**field** ((*vector number number*) -> (*vector number number*)) *field-options∗*)
(*contour* (*number number -> number*) *contour-options∗*)
(*shade* (*number number -> number*) *shade-options∗*)
(*mix 2d-plot-item 2d-plot-item+*)
(*custom* (*2d-view\% ->* `void`))

3d-plot-item is one of:
(*surface* (*number number -> number*) *surface-options∗*)
(*mix 3d-plot-item 3d-plot-item+*)
(*custom* (*3d-view\% ->* `void`))

note: all of the options appear as
option-name **:** *enumeration of* `values` **with** *default enclosed in* []
or
option-name **:** *type* [*default*]

color is one of:
'white 'black 'yellow 'green 'aqua 'pink
'wheat 'grey 'brown 'blue 'violet 'cyan
'turquoise 'magenta 'salmon 'red

point-options are:
sym **:** ['`square`], 'circle, 'odot, 'bullet
color **:** *color* ['black]

line-options are:
samples **:** *number* [150]
width **:** *number* [1]
color **:** *color* ['red]
mode **:** ['standard], 'parametric
mapping **:** ['cartesian], 'polar
t-min **:** *number* [−5]
t-max **:** *number* [5]

error-bar-options are:
color **:** *color* ['red]

field-options are:
color **:** *color* ['red]
width **:** *number* [1]
style **:** ['scaled],'normalized,'`read`

contour-options are:
samples **:** *number* [50]
color **:** *color* ['black]
width **:** *number* [1]
levels **:** *number U* (*list-of number*) [10]

shade-options are:
samples **:** *number* [50]
levels **:** *number* [10]

surface-options are:
samples **:** *number* [50]
color **:** *color* ['black]
width **:** *number* [1]

The 2d and 3d plot-items can be created in several ways. The first is by using the built-in constructors with your own data.

points will draw points on a graph given a list of vectors specifying their location. Sym specifies the appearance of the points.

line will draw a line specified in either functional, ie. y=f(x), or parametric mode, x,y = f(t). If the function is parametric, the line-option *mode* must be set to *parametric*. *t-min* and *t-max* set the parameter when in parametric mode. *mapping* can be set to 'radial.

error-bars will draw error bars given a list of vectors. The vector specifies the center of the error bar (x,y) as the first two elements, and its magnitude as the third.

field will draw a vector field from a vector valued function. Styles are either *real*, *scaled*, or *normalized*.

Both *shade* and *contour* will render 3d functions on a 2d graph using colored shades and contours (respectively) to represent the value of the function at that position. *contour* will let you choose the levels explicitly if desired, by setting the *levels* option to a list of contour levels to be plotted.

surface plots a 3d surface in a 3d box, showing only the *top* of the surface.

2.2 Curve Fitting

PLTPlot uses the standard Non-Linear Least Squares fit algorithm for curve fitting. The code that implements the algorithm is public domain, and is used by the gnuplot package.

Data:

a fit-result is
(*fit* (*number*∗-> *number*) *parameter-guess-list data*)

parameter-guess-list is a set of name-value pairs enclosed in (..)

data is
(*list-of* (*vector number number number*))
| (*list-of* (*vector number number number number*))

Functions:

fit-result-function **:** *fit-result -> procedure*
fit-result-final-params **:** *fit-result -> guess-list*

The *fit* form attempts to fit a a *fittable-function* to the data that is given. The *guess-list*should be set of parameters and values. The more accurate your initial guesses are, the more likely the fit is to succeed. If there are no good values for the guesses, leave them as 1.

fit-result-final-params returns an associative list of the parameters specified in fit and their values. Note that the values may not be correct if the fit failed to converge. For a visual test use *fit-result-function* to get the function with the parameters in place and plot it along with the original data.

2.3 Misc Functions

The plot library comes with a few useful miscellaneous functions:

derivative **:** (*number -> number*) [*h* .000001] -> (*number -> number*)
gradient **:** (*number number -> number*) [*h* .000001] -> (*vector -> vector*)
make-vec **:** (*number number -> number*) (*number number -> number*) -> (*vector -> vector*)

derivative creates a function that evaluates the numeric derivative of the given single-variable function using the definition. *h* is the divisor used in the calculation.

gradient creates a vector-valued function that is the gradient of the given function. *h* represents the numeric divisor, as with the derivative function.

make-vec creates one vector valued function from two parts.

3. Module: plot-extend.ss

plot-extend.ss allows you to create your own constructors, further customize the appearance of the plot windows, and in general extend the package.

3.1 `2d-view%`

Provides an interface to drawing 2dplots. Some methods call low-level functions while others are emulated in scheme.

- *set-labels* **:** *string string string* -> `void`
  Sets x, y and title labels
- *plot-vector* **:** *vector vector* -> `void`
  Plots a single vector. First argument is the head, second is the tail.
- *plot-vectors* **:** (**listof** (`list` *vector vector*)) -> `void`
  Plots a list of vectors. Each vector is a list of two scheme *vector*s.
- *plot-points* **:** (**listof** *vector*) *number* ->*void*
  Plots points using a specified character. ** provide character map **
- *plot-line* **:** (**listof** (*vector number number*)) -> `void`
  Plots a line given a set of points. Each point is represented by a *vector*.
- *plot-contours* **:** (**listof** (*lisftof number*)) (**listof** *number*) (**listof** *number*) (**listof** *number*) ->*void*
  Plots a grid representing a 3d function using contours to distinguish levels. Args are grid, xvalues yvalues and levels to plot.
- *plot-shades* **:** (**listof** (*lisftof number*)) (**listof** *number*) (**listof** *number*) (**listof** *number*) ->*void*
  Plots a grid representing a 3d function using shades to represent height (z).

3.2 `3d-view%`

Provides an interface to drawing 3d plots.

- *plot-surface* **:** (**listof** (*lisftof number*)) (**listof** *number*) (**listof** *number*) ->*void*
  Plots a grid representing a 3d function in a 3D box, showing only the top of the surface.
- *plot-line* **:** (**listof** *number*) (**listof** *number*) (**listof** *number*) -> `void`
  Plots a line in 3d space. The arguments are lists of x,y and z coordinates respectively.
- *get-z-min* **:** -> *number*
  Returns the minimum plottable Z coordinate.

- *get-z-max* **:** -> *number*
  Returns the maximum plottable Z coordinate.

- *get-alt* **:** -> *number*
  Returns the altitude (in degrees) from which the 3d box is viewed.

- *get-az* **:** -> *number*
  Returns the azimuthal angle.

3.3 define-plot-type

Macro used to create new constructors. It is easiest to explain with an example, so here is an implementation of a simple line constructor:

```
(define-plot-type line
  func 2dplotview (x-min x-max) ((samples 150) (color 'red) (width 1))
  (send* 2dplotview
    (set-line-color color) (set-line-width width)
    (plot-line (map (lambda (x) (vector x (func x)))
                    (x-values samples x-min x-max)))))
```

- The first keyword after then name of the new plot type, is used to refer to the data that will be rendered. In this case, we will are calling our data *func*. For example, in the execution of (*plot* (*line* (**lambda** (*x*) *x*))) *func* would refer to the indentify function.

- *2dplotview* refers to the name of the view object that the Plot-item will be applied to by *plot*

- The *x-min* and *x-max* are fields in the *2d-view\%* object. They will be bound to the values of those fields before the execution of the body, assuming the object has the methods *get-x-min* and *get-x-max*. This entire expression can be omitted if none of the fields are necessary (such as for plotting discrete data points).

- The last set of parenthesized expressions sets keywords based arguments and their default values for the constructor. To over-ride values the user needs to provide an associative list with the desired values. Ex: (*line* (**lambda** (*x*) *x*) '((color blue)))

Index