PLoT: Graph Plotting

Version 5.1.1

Alexander Friedman
and Jamie Raymond

April 30, 2011

PLoT (a.k.a. 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.

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 plot)` try

```
(plot (line (lambda (x) x)))
```

Any other function using the contract `(-> real? real?)` 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 brackets 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 argument/value pairs after the data.

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

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

- Discrete data, such as

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

3

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 (vector-field (gradient 3dfun)))
```

or in a 3d box

- – Displaying only the top of the surface

```
(plot3d (surface 3dfun))
```

1.3   Curve Fitting

The `plot` library uses a non-linear, least-squares fit algorithm to fit parameterized functions to given data.

To fit a particular function to a curve:

- Set up the independent and dependent variable data. The first item in each vector is the independent variable, the second is the result. The last item 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)))
```

4

- 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))))
```

- 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.

- Do the fit; the details of the function are described in §2.2 "Curve Fitting".

```
(define fitted
  (fit fit-fun
       '((m 1) (b 1))
       data))
```

- View the resulting parameters; for example,

```
(fit-result-final-params fitted)
```

will produce (2.0 3.0).

- 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 the function.

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

A more realistic example can be found in "demos/fit-demo-2.ss" in the "plot" collection.

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 2-D and 3-D 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 §3 "Customizing Plots".

For example, if we wanted to create a constructor that creates plot-items that draw dashed lines given a (-> real? real?) function, we could do the following:

```
(require plot/extend)
```

5

```
(define (dashed-line fun
                     #:x-min [x-min -5]
                     #:x-max [x-max 5]
                     #:samples [samples 100]
                     #:segments [segments 20]
                     #:color [color 'red]
                     #:width [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)))))))
    (lambda (2dview)
      (send 2dview set-line-color color)
      (send 2dview 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 using `dashed-line`:

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

2   Plotting

```
(require plot)
```

The `plot` library provides the ability to make basic plots, fit curves to data, and some useful miscellaneous functions.

2.1   Plotting

The `plot` and `plot3d` functions generate plots that can be viewed in the DrRacket interactions window.

```
(plot  data
      [#:width width
       #:height height
       #:x-min x-min
       #:x-max x-max
       #:y-min y-min
       #:y-max y-max
       #:x-label x-label
       #:y-label y-label
       #:title title
       #:fgcolor fgcolor
       #:bgcolor bgcolor
       #:lncolor lncolor
       #:out-file out-file]) → (is-a?/c snip%)
  data : ((is-a?/c 2d-view%) . -> . void?)
  width : real? = 400
  height : real? = 400
  x-min : real? = -5
  x-max : real? = 5
  y-min : real? = -5
  y-max : real? = 5
  x-label : string? = "X axis"
  y-label : string? = "Y axis"
  title : string? = ""
  fgcolor : (list/c byte? byte? byte) = '(0 0 0)
  bgcolor : (list/c byte? byte? byte) = '(255 255 255)
  lncolor : (list/c byte? byte? byte) = '(255 0 0)
  out-file : (or/c path-string? output-port? #f) = #f
```

Plots `data` in 2-D, where `data` is generated by functions like `points` or `lines`.

A `data` value is represented as a procedure that takes a `2d-view%` instance and adds plot information to it.

The result is a `snip%` for the plot. If an `#:out-file` path or port is provided, the plot is also written as a PNG image to the given path or port.

```
(plot3d  data
         [#:width width
          #:height height
          #:x-min x-min
          #:x-max x-max
          #:y-min y-min
          #:y-max y-max
          #:z-min z-min
          #:z-max z-max
          #:alt alt
          #:az az
          #:x-label x-label
          #:y-label y-label
          #:z-label z-label
          #:title title
          #:fgcolor fgcolor
          #:bgcolor bgcolor
          #:lncolor lncolor]) → (is-a?/c snip%)
  data : ((is-a?/c 3d-view%) . -> . void?)
  width : real? = 400
  height : real? = 400
  x-min : real? = -5
  x-max : real? = 5
  y-min : real? = -5
  y-max : real? = 5
  z-min : real? = -5
  z-max : real? = 5
  alt : real? = 30
  az : real? = 45
  x-label : string? = "X axis"
  y-label : string? = "Y axis"
  z-label : string? = "Z axis"
  title : string? = ""
  fgcolor : (list/c byte? byte? byte) = '(0 0 0)
  bgcolor : (list/c byte? byte? byte) = '(255 255 255)
  lncolor : (list/c byte? byte? byte) = '(255 0 0)
```

Plots `data` in 3-D, where `data` is generated by a function like `surface`. The arguments `alt` and `az` set the viewing altitude (in degrees) and the azimuth (also in degrees), respectively.

A 3-D `data` value is represented as a procedure that takes a `3d-view%` instance and adds plot information to it.

```
(points vecs [#:sym sym #:color color])
 → ((is-a?/c 2d-view%) . -> . void?)
  vecs : (listof (vector/c real? real?))
  sym : (or/c character? integer? symbol?) = 'fullsquare
  color : plot-color? = 'black
```

Creates 2-D plot data (to be provided to `plot`) given a list of points specifying locations. The `sym` argument determines the appearance of the points. It can be a symbol, an ASCII character, or a small integer (between -1 and 127). The following symbols are known: 'pixel, 'dot, 'plus, 'asterisk, 'circle, 'times, 'square, 'triangle, 'oplus, 'odot, 'diamond, '5star, '6star, 'fullsquare, 'bullet, 'full5star, 'circle1, 'circle2, 'circle3, 'circle4, 'circle5, 'circle6, 'circle7, 'circle8, 'left-arrow, 'rightarrow, 'uparrow, 'downarrow.

```
(line  f
      [#:samples samples
       #:width width
       #:color color
       #:mode mode
       #:mapping mapping
       #:t-min t-min
       #:t-max t-min])   → ((is-a?/c 2d-view%) . -> . void?)
  f : (real? . -> . (or/c real? (vector real? real?)))
  samples : exact-nonnegative-integer? = 150
  width : exact-positive-integer? = 1
  color : plot-color? = 'red
  mode : (one-of/c 'standard 'parametric) = 'standard
  mapping : (or-of/c 'cartesian 'polar) = 'cartesian
  t-min : real? = -5
  t-min : real? = 5
```

Creates 2-D plot data to draw a line.

The line is specified in either functional, i.e. $y = f(x)$, or parametric, i.e. $x,y = f(t)$, mode. If the function is parametric, the `mode` argument must be set to `'parametric`. The `t-min` and `t-max` arguments set the parameter when in parametric mode.

```
(error-bars vecs [#:color color])
 → ((is-a?/c 2d-view%) . -> . void?)
  vecs : (listof (vector/c real? real? real?))
  color : plot-color? = 'black
```

Creates 2-D plot data for error bars given a list of vectors. Each vector specifies the center of the error bar (*x,y*) as the first two elements and its magnitude as the third.

```
(vector-field  f
               [#:width width
                #:color color
                #:style style])
 → ((is-a?/c 2d-view%) . -> . void?)
  f : ((vector real? real?) . -> . (vector real? real?))
  width : exact-positive-integer? = 1
  color : plot-color? = 'red
  style : (one-of/c 'scaled 'normalized 'read) = 'scaled
```

Creates 2-D plot data to draw a vector-field from a vector-valued function.

```
(contour  f
          [#:samples samples
           #:width width
           #:color color
           #:levels levels]) → ((is-a?/c 2d-view%) . -> . void?)
  f : (real? real? . -> . real?)
  samples : exact-nonnegative-integer? = 50
  width : exact-positive-integer? = 1
  color : plot-color? = 'black
  levels : (or/c exact-nonnegative-integer? = 10
                 (listof real?))
```

Creates 2-D plot data to draw contour lines, rendering a 3-D function a 2-D graph cotours (respectively) to represent the value of the function at that position.

```
(shade f [#:samples samples #:levels levels])
 → ((is-a?/c 2d-view%) . -> . void?)
  f : (real? real? . -> . real?)
  samples : exact-nonnegative-integer? = 50
  levels : (or/c exact-nonnegative-integer? = 10
                 (listof real?))
```

Creates 2-D plot data to draw like contour, except using shading instead of contour lines.

```
(surface  f
          [#:samples samples
           #:width width
           #:color color])    → ((is-a?/c 3d-view%) . -> . void?)
```

```
f : (real? real? . -> . real?)
samples : exact-nonnegative-integer? = 50
width : exact-positive-integer? = 1
color : plot-color? = 'black
```

Creates 3-D plot data to draw a 3-D surface in a 2-D box, showing only the *top* of the surface.

```
(mix data ...+) → (any/c . -> . void?)
  data : (any/c . -> . void?)
```

Creates a procedure that calls each `data` on its argument in order. Thus, this function can composes multiple plot `data`s into a single data.

```
(plot-color? v) → boolean?
  v : any/c
```

Returns `#t` if `v` is one of the following symbols, `#f` otherwise:

```
'white 'black 'yellow 'green 'aqua 'pink
'wheat 'grey 'blown 'blue 'violet 'cyan
'turquoise 'magenta 'salmon 'red
```

2.2   Curve Fitting

PLoT 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.

```
(fit f guess-list data) → fit-result?
  f : (real? ... . -> . real?)
  guess-list : (list/c (list symbol? real?))
  data : (or/c (list-of (vector/c real? real? real?))
               (list-of (vector/c real? real? real? real?)))
```

Attempts to fit a *fittable function* to the data that is given. The `guess-list` should be a set of arguments 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.

11

```
(struct fit-result (rms
                    variance
                    names
                    final-params
                    std-error
                    std-error-percent
                    function)
        #:extra-constructor-name make-fit-result)
  rms : real?
  variance : real?
  names : (listof symbol?)
  final-params : (listof real?)
  std-error : (listof real?)
  std-error-percent : (listof real?)
  function : (real? ... . -> . real?)
```

The `params` field contains 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 the `function` field to get the function with the parameters in place and plot it along with the original data.

2.3   Miscellaneous Functions

```
(derivative f [h]) → (real? . -> . real?)
  f : (real? . -> . real?)
  h : real? = 1e-06
```

Creates a function that evaluates the numeric derivative of `f`. The given `h` is the divisor used in the calculation.

```
(gradient f [h])
 → ((vector/c real? real?) . -> . (vector/c real? real?))
  f : (real? real? . -> . real?)
  h : real? = 1e-06
```

Creates a vector-valued function that the numeric gradient of `f`.

```
(make-vec fx fy)
 → ((vector/c real? real?) . -> . (vector/c real? real?))
  fx : (real? real? . -> . real?)
  fy : (real? real? . -> . real?)
```

Creates a vector-values function from two parts.

3   Customizing Plots

```
(require plot/extend)
```

The `plot/extend` module allows you to create your own constructors, further customize the appearance of the plot windows, and in general extend the package.

```
(sample-size sample-count x-min x-max) → real?
  sample-count : exact-positive-integer?
  x-min : number
  x-max : number
```

Given *sample-count*, *x-min*, and *x-max*, returns the size of each sample.

```
(scale-vectors vecs
               x-sample-size
               y-sample-size) → (listof vector?)
  vecs : (listof vector?)
  x-sample-size : real?
  y-sample-size : real?
```

Scales vectors, causing them to fit in their boxes.

```
(x-values sample-count x-min x-max) → (listof real?)
  sample-count : exact-positive-integer?
  x-min : number
  x-max : number
```

Given `samples`, *x-min*, and *x-max*, returns a list of `xs` spread across the range.

```
(normalize-vector vec
                  x-sample-size
                  y-sample-size) → vector?
  vec : vector?
  x-sample-size : real?
  y-sample-size : real?
```

Normalizes *vec* based on *x-sample-size* and *y-sample-size*.

```
(normalize-vectors vecs
                   x-sample-size
                   y-sample-size) → (listof vector?)
```

```
vecs : (listof vector?)
x-sample-size : real?
y-sample-size : real?
```

Normalizes *vecs* based on *x-sample-size* and *y-sample-size*.

```
(make-column x ys) → (listof (vector/c real? real?))
  x : real?
  ys : (listof real?)
```

Given an *x* and a list of *y*s, produces a list of points pairing the *x* with each of the *y*s.

```
(xy-list sample-count x-min x-max y-min y-max)
 → (listof (listof (vector/c real? real?)))
  sample-count : exact-positive-integer?
  x-min : real?
  x-max : real?
  y-min : real?
  y-max : real?
```

Makes a list of all the positions on the graph.

```
(zgrid f xs ys) → (listof (listof real?))
  f : (real? real? . -> . real?)
  xs : (listof real?)
  ys : (listof real?)
```

Given a function that consumes *x* and *y* to produce *z*, a list of *x*s, and a list of *y*s, produces a list of *z* column values.

```
plot-view% : class?
  superclass: object%
```

```
(send a-plot-view get-x-min) → real?
```
     Returns the minimum plottable *x* coordinate.

```
(send a-plot-view get-y-min) → real?
```
     Returns the minimum plottable *y* coordinate.

```
(send a-plot-view get-x-max) → real?
```
     Returns the maximum plottable *x* coordinate.

```
(send a-plot-view get-y-max) → real?
```
Returns the maximum plottable *y* coordinate.

```
(send a-plot-view set-line-color color) → void?
  color : plot-color?
```
Sets the drawing color.

```
(send a-plot-view set-line-width width) → void?
  width : real?
```
Sets the drawing line width.

```
2d-view% : class?
  superclass: plot-view%
```

Provides an interface to drawing 2-D plots. An instance of `2d-view%` is created by `plot`, and the following methods can be used to adjust it.

```
(send a-2d-view set-labels x-label
                           y-label
                           title)  → void?
  x-label : string?
  y-label : string?
  title : string?
```
Sets the axis labels and title.

```
(send a-2d-view plot-vector head tail) → void?
  head : (vector/c real? real?)
  tail : (vector/c real? real?)
```
Plots a single vector.

```
(send a-2d-view plot-vectors vecs) → void?
  vecs : (listof (list/c (vector/c real? real?)
                         (vector/c real? real?)))
```
Plots a set of vectors.

```
(send a-2d-view plot-points points sym) → void?
  points : (listof (vector/c real? real?))
  sym : (or/c character? integer? symbol?)
```
Plots points using a specified symbol. See *points* for possible values for *sym*

```
(send a-2d-view plot-line points) → void?
  points : (listof (vector/c real? real?))
```

     Plots a line given a set of points.

```
(send a-2d-view plot-contours grid
                              xs
                              ys
                              levels) → void?
  grid : (listof (listof real?))
  xs : (listof real?)
  ys : (listof real?)
  levels : (listof real?)
```

     Plots a grid representing a 3-D function using contours to distinguish levels.

```
(send a-2d-view plot-shades grid
                            xs
                            ys
                            levels) → void?
  grid : (listof (listof real?))
  xs : (listof real?)
  ys : (listof real?)
  levels : (listof real?)
```

     Plots a grid representing a 3-D function using shades to show levels.

```
3d-view% : class?
  superclass: plot-view%
```

Provides an interface to drawing 3-D plots. An instance of `3d-view%` is created by `plot3d`, and the following methods can be used to adjust it.

```
(send a-3d-view plot-surface xs ys zs) → void?
  xs : (listof real?)
  ys : (listof real?)
  zs : (listof real?)
```

     Plots a grid representing a 3d function in a 3d box, showing only the top of the surface.

```
(send a-3d-view plot-line xs ys zs) → void?
  xs : (listof real?)
  ys : (listof real?)
```

zs : (listof real?)

> Plots a line in 3-D space.

(send *a-3d-view* get-z-min) → real?

> Returns the minimum plottable z coordinate.

(send *a-3d-view* get-z-max) → real?

> Returns the maximum plottable z coordinate.

(send *a-3d-view* get-alt) → real?

> Returns the altitude (in degrees) from which the 3-D box is viewed.

(send *a-3d-view* get-az) → real?

> Returns the azimuthal angle.