PLoT: Graph Plotting

Version 5.2

Neil Toronto <neil.toronto@gmail.com>

November 8, 2011

```
(require plot)
```

PLoT provides a flexible interface for producing nearly any kind of plot. It includes many common kinds already, such as scatter plots, line plots, contour plots, histograms, and 3D surfaces and isosurfaces. Thanks to Racket's excellent multiple-backend drawing library, PLoT can render plots as manipulatable images in DrRacket, as bitmaps in slideshows, as PNG, PDF, PS and SVG files, or on any device context.

A note on backward compatibility. PLoT has undergone a major rewrite between versions 5.1.3 and 5.2. Many programs written using PLoT 5.1.3 and earlier will still compile, run and generate plots. Some programs will not. Most programs use deprecated functions such as `mix`, `line` and `surface`. These functions still exist for backward compatibility, but are deprecated and may be removed in the future. If you have code written for PLoT 5.1.3 or earlier, please see §10 "Porting From PLoT <= 5.1.3" (and possibly §11 "Compatibility Module").

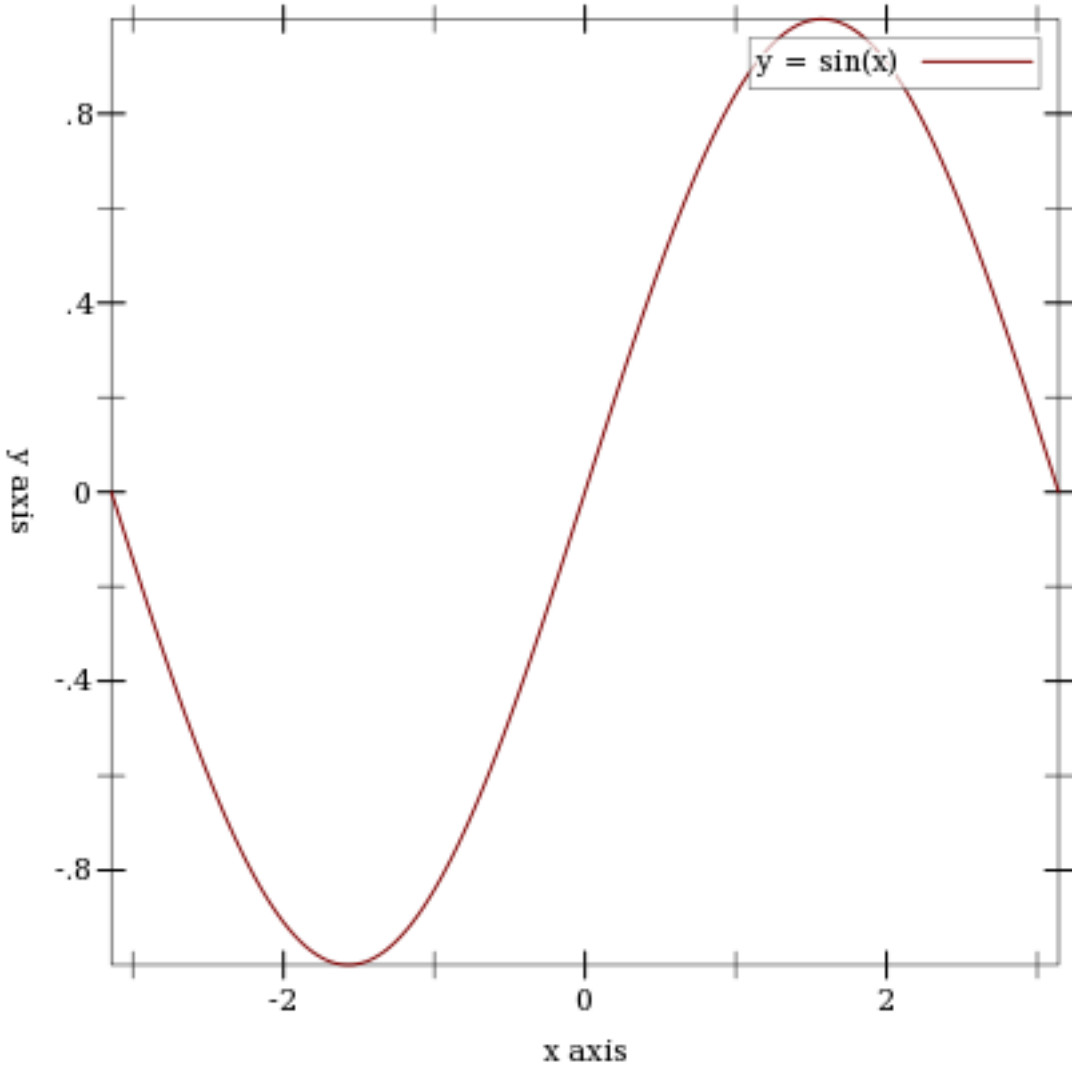
Contents

1 Introduction

1.1 Plotting 2D Graphs

To plot a one-input, real-valued function, first `(require plot)`, and then try something like

```
> (plot (function sin (- pi) pi #:label "y = sin(x)"))
```



The first argument to `function` is the function to be plotted, and the `#:label` argument becomes the name of the function in the legend.

1.2 Terminology

In the above example, `(- pi)` and `pi` define the *x*-axis *bounds*, or the closed interval in which to plot the `sin` function. The `function` function automatically determines that the *y*-axis bounds should be [-1,1].

The `function` function constructs a *renderer*, which does the actual drawing. A renderer also produces legend entries, requests bounds to draw in, and requests axis ticks and tick labels.

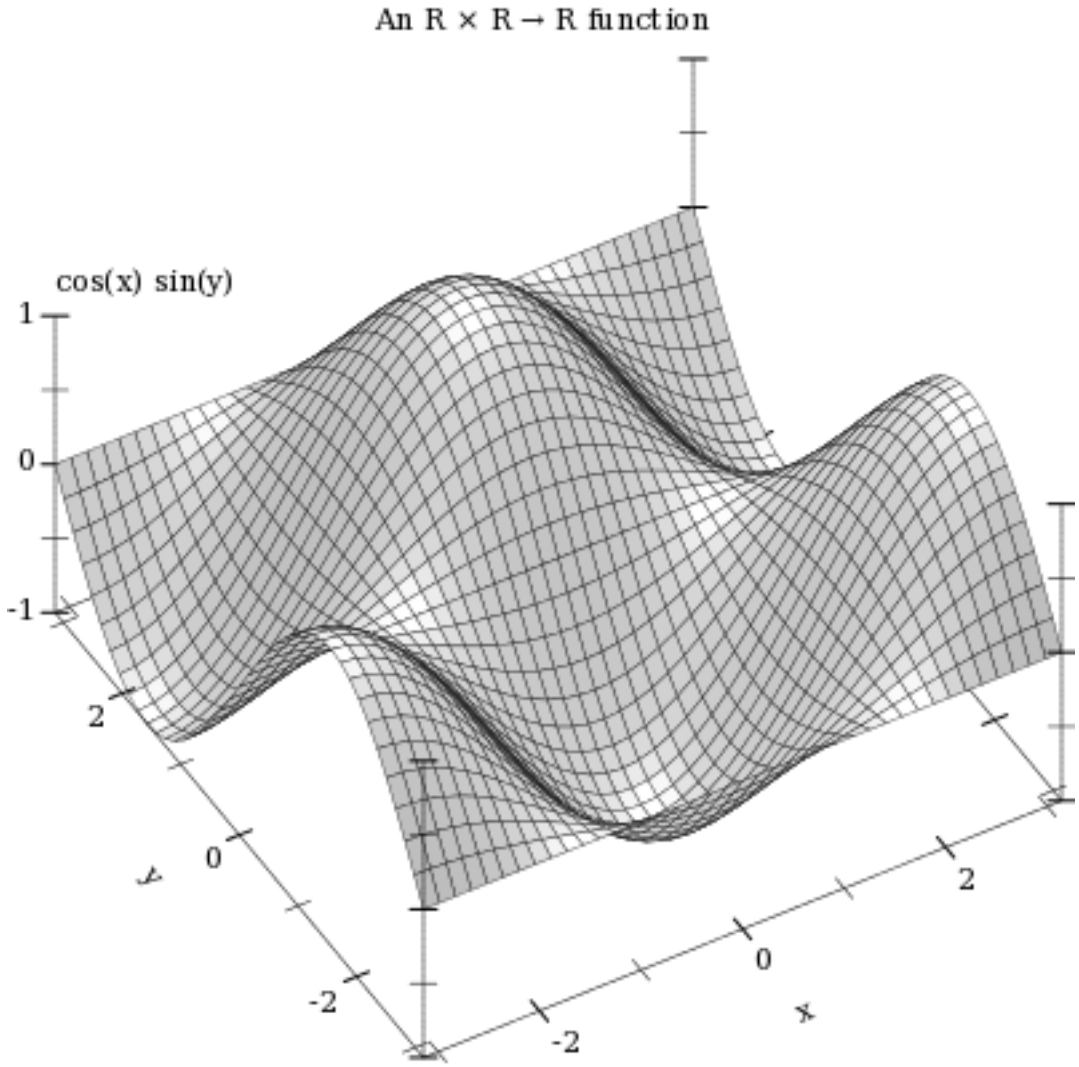
The `plot` function collects legend entries, bounds and ticks. It then sets up a *plot area* with large enough bounds to contain the renderers, draws the axes and ticks, invokes the renderers' drawing procedures, and then draws the legend.

1.3 Plotting 3D Graphs

To plot a two-input, real-valued function as a surface, first `(require plot)`, and then try something like

```
> (plot3d (surface3d (λ (x y) (* (cos x) (sin y)))
                     (- pi) pi (- pi) pi)
          #:title "An R × R ⟶ R function"
          #:x-label "x" #:y-label "y" #:z-label "cos(x) sin(y)")
```

The documentation can't show it, but in DrRacket you can rotate 3D plots by clicking on them and dragging the mouse. Try it!

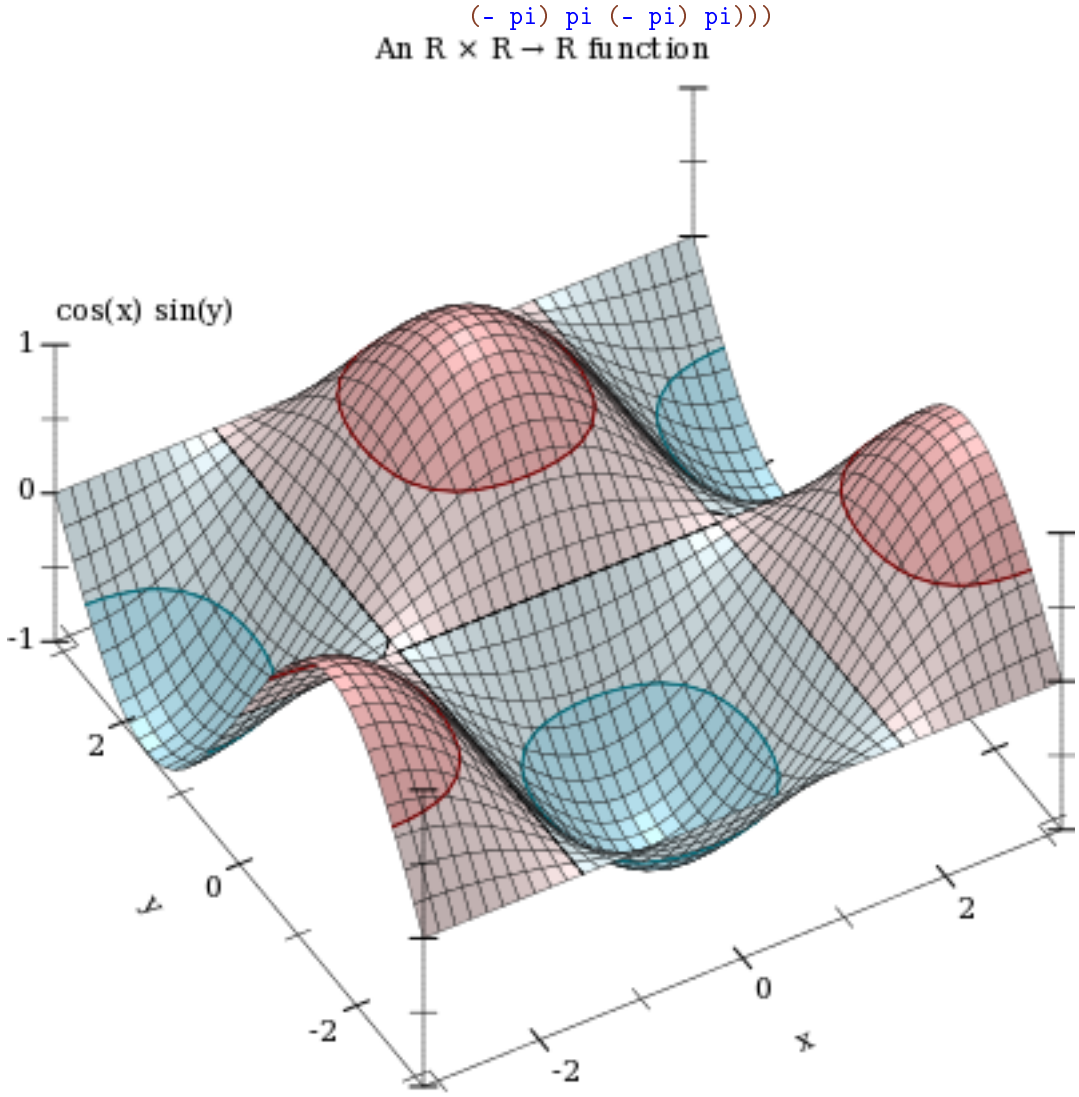


This example also demonstrates using keyword arguments that change the plot, such as `#:title`. In PLoT, every keyword argument is optional and almost all have parameterized default values. In the case of `plot3d`'s `#:title`, the corresponding parameter is `plot-title`. That is, keyword arguments are usually shortcuts for parameterizing plots or renderers:

```
> (parameterize ([plot-title  "An R × R → R function"]
                 [plot-x-label "x"]
                 [plot-y-label "y"]
                 [plot-z-label "cos(x) sin(y)"])
    (plot3d (contour-intervals3d (λ (x y) (* (cos x) (sin y)))
```

When parameterizing more than one plot, it is often easier to set parameters globally, as in `(plot-title "Untitled")` and `(plot3d-angle 45)`. There are many parameters that do not correspond to keyword arguments, such as `plot-font-size`. See §7 "Plot and Renderer Parameters" for the full listing.

```
                                (- pi) pi (- pi) pi)))
```



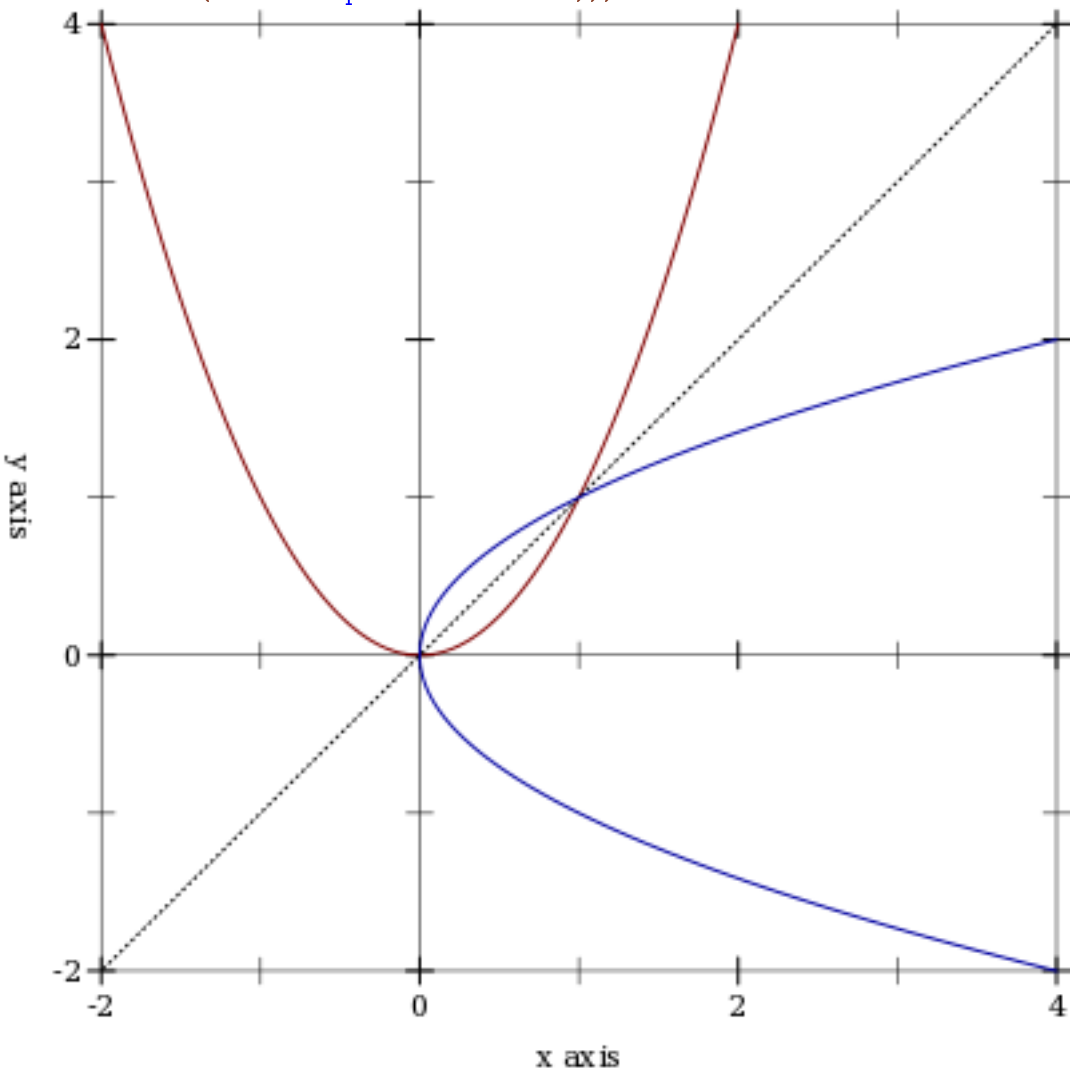
This example also demonstrates `contour-intervals3d`, which colors the surface between contour lines, or lines of constant height. By default, `contour-intervals3d` places the contour lines at the same heights as the ticks on the z axis.

1.4 Plotting Multiple 2D Renderers

Renderers may be plotted together by passing them in a list:

```
> (plot (list (axes)
```

```
(function sqr -2 2)
(function (λ (x) x) #:color 0 #:style 'dot)
(inverse sqr -2 2 #:color 3)))
```



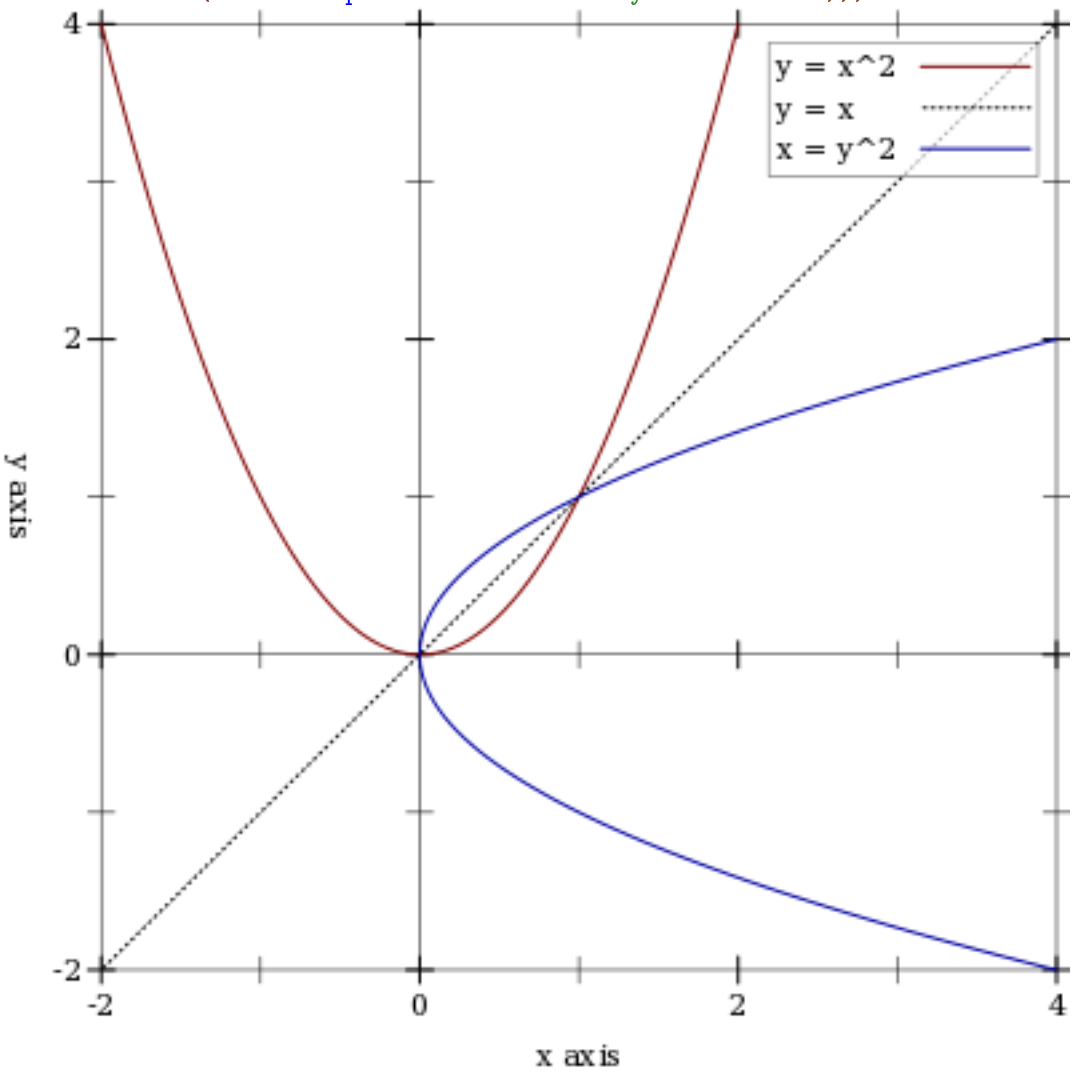
Here, `inverse` plots the inverse of a function. (Both `function` and `inverse` plot the reflection line `(λ (x) x)` identically.)

Notice the numbered colors. PLoT additionally recognizes, as colors, lists of RGB values such as `'(128 128 0)`, `color%` instances, and strings like `"red"` and `"navajowhite"`. (The last are turned into RGB triples using a `color-database<%>`.) Use numbered colors when you just need different colors with good contrast, but don't particularly care what they are.

The `axes` function returns a list of two renderers, one for each axis. This list is passed in a list to `plot`, meaning that `plot` accepts *lists of lists* of renderers. In general, both `plot` and `plot3d` accept a `treeof` renderers.

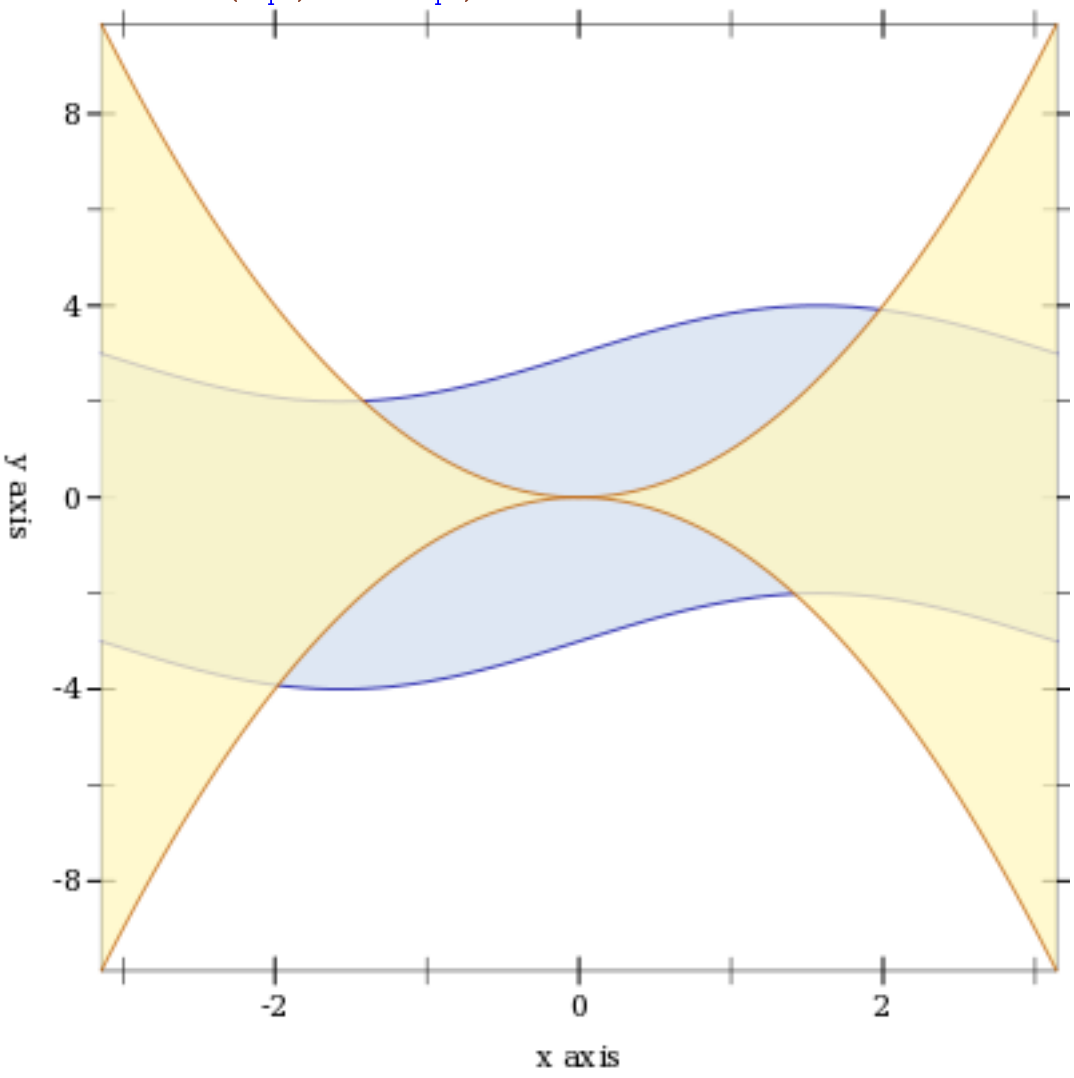
Renderers generate legend entries when passed a `#:label` argument. For example,

```
> (plot (list (axes)
              (function sqr -2 2 #:label "y = x^2")
              (function (λ (x) x) #:label "y =
x" #:color 0 #:style 'dot)
              (inverse sqr -2 2 #:label "x = y^2" #:color 3)))
```



Lists of renderers are `flatten`ed, and then plotted *in order*. The order is more obvious with interval plots:

```
> (plot (list (function-interval (λ (x) (- (sin x) 3))
                                 (λ (x) (+ (sin x) 3)))
              (function-interval (λ (x) (- (sqr x))) sqr #:color 4
                                 #:line1-color 4 #:line2-color 4))
        #:x-min (- pi) #:x-max pi)
```



Clearly, the blue-colored interval between sine waves is drawn first.

1.5 Renderer and Plot Bounds

In the preceeding example, the x-axis bounds are passed to `plot` using the keyword arguments `#:x-min` and `x-max`. The bounds could easily have been passed in either call to `function-interval` instead. In both cases, `plot` and `function-interval` work together to determine y-axis bounds large enough for both renderers.

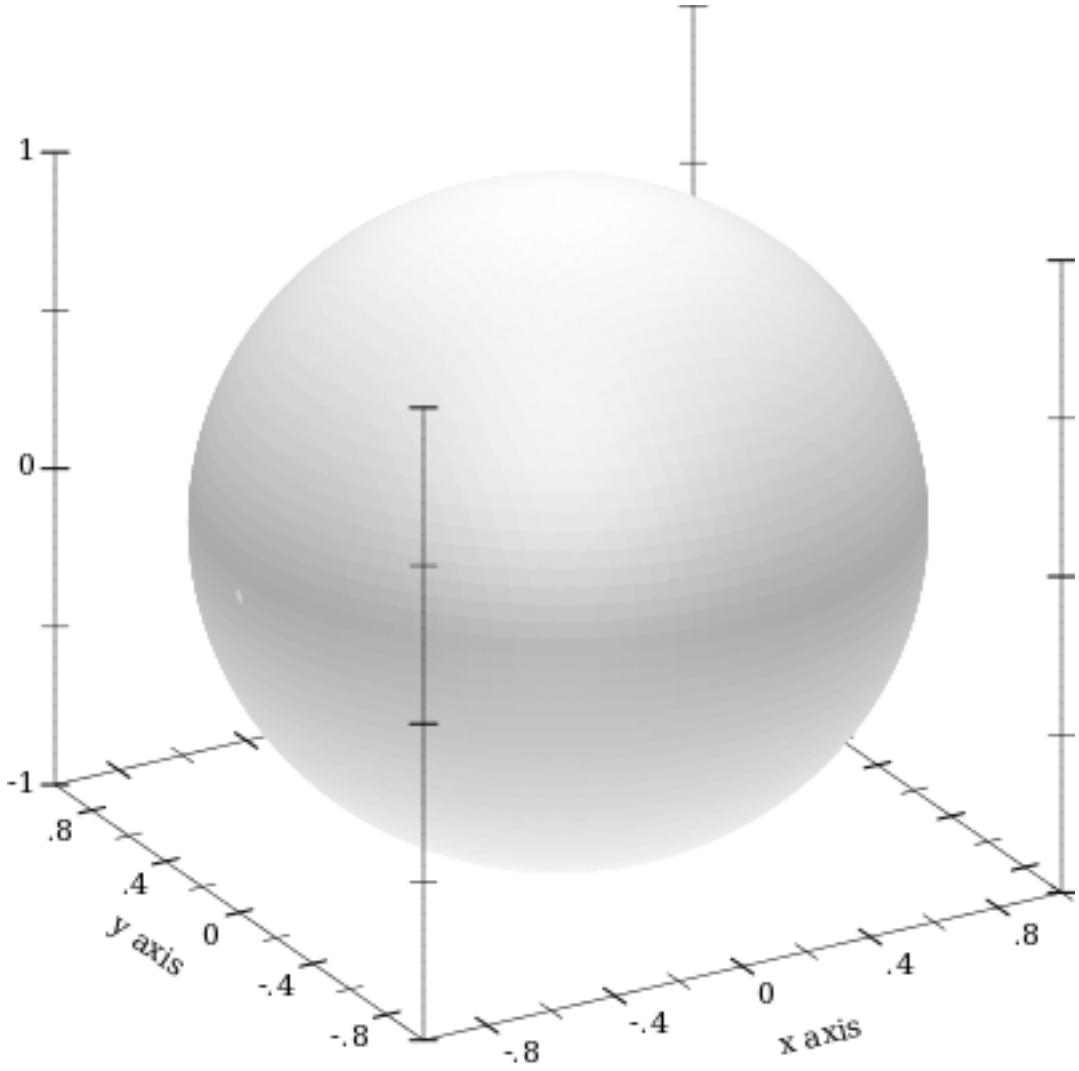
It is not always possible for renderers and `plot` or `plot3d` to determine the bounds:

```
> (plot (function sqr))
plot: could not determine nonempty x axis; got: x-min = #f,
x-max = #f
> (plot (function sqr #f #f))
plot: could not determine nonempty x axis; got: x-min = #f,
x-max = #f
> (plot (function sqr (- pi)))
plot: could not determine nonempty x axis; got: x-min =
-3.141592653589793, x-max = #f
> (plot (list (function sqr #f 0)
              (function sqr 0 #f)))
plot: could not determine nonempty x axis; got: x-min = 0,
x-max = 0
```

There is a difference between passing bounds to renderers and passing bounds to `plot` or `plot3d`: bounds passed to `plot` or `plot3d` cannot be changed by a renderer that requests different bounds. We might say that bounds passed to renderers are *suggestions*, and bounds passed to `plot` and `plot3d` are *commandments*.

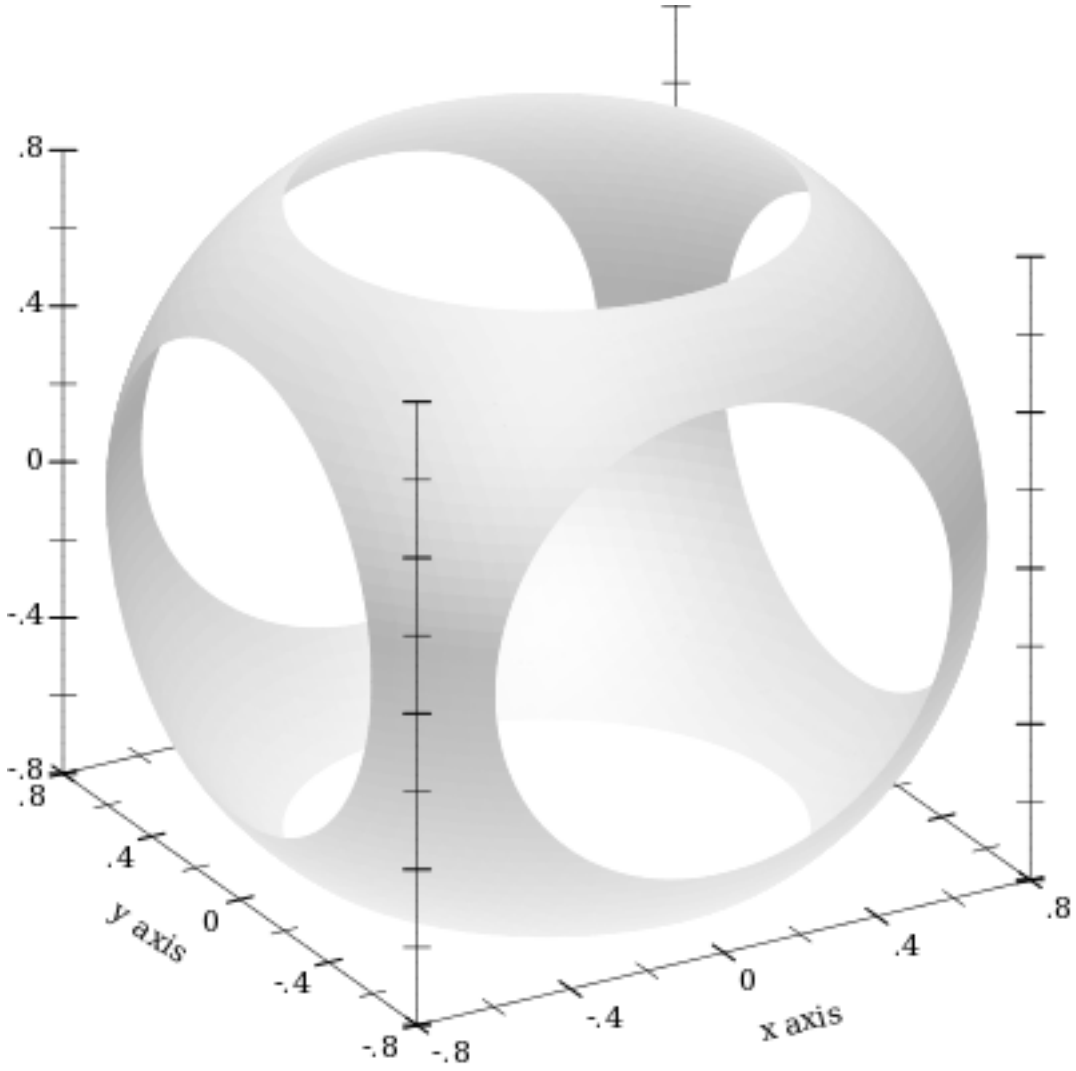
Here is an example of commanding `plot3d` to override a renderer's bounds. First, consider the plot of a sphere with radius `1`:

```
> (plot3d (polar3d (λ (θ ρ) 1) #:line-color "white" #:line-
width 1)
          #:altitude 25)
```



Passing bounds to `plot3d` that are smaller than [-1..1] × [-1..1] × [-1..1] cuts off the six axial poles:

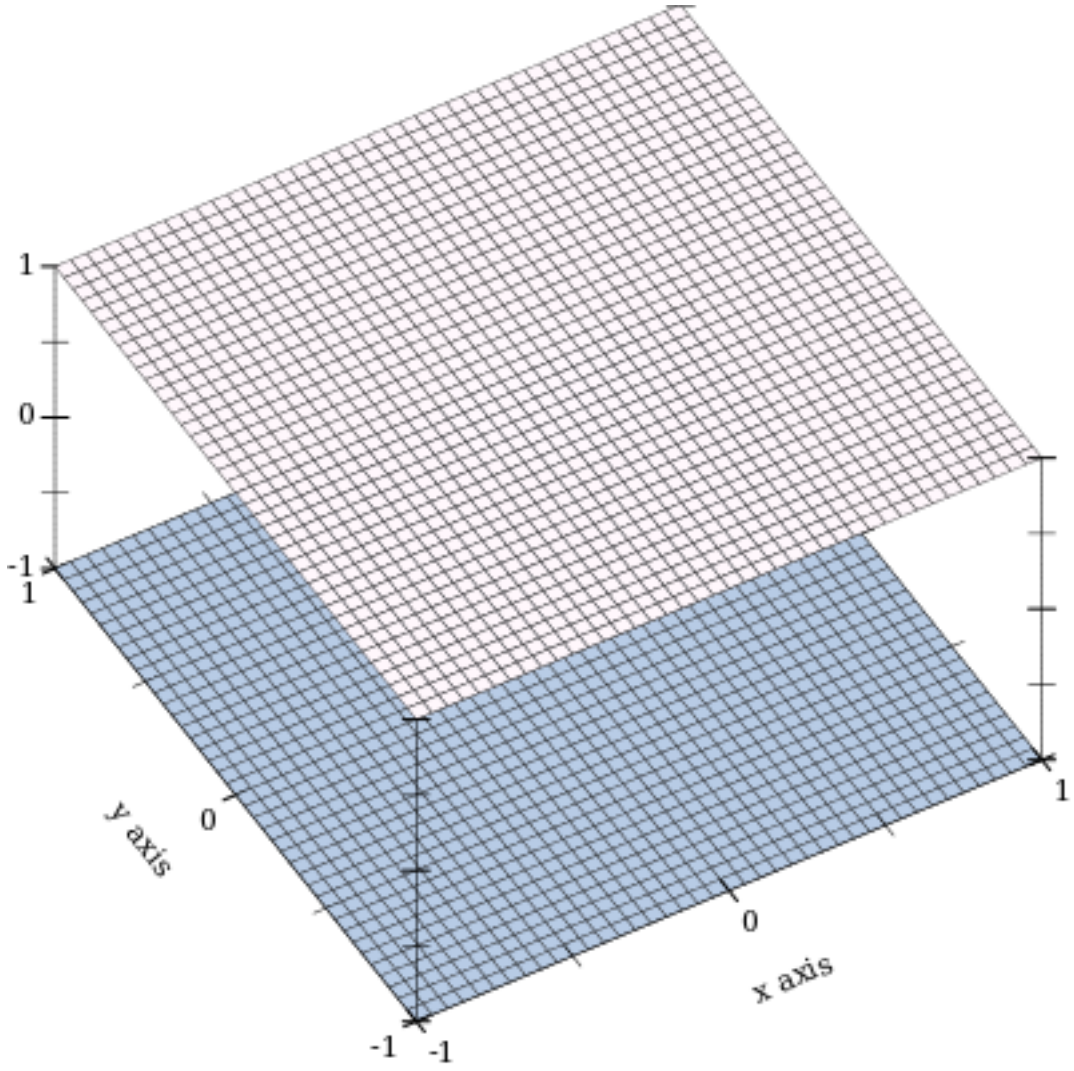
```
> (plot3d (polar3d (λ (θ ρ) 1) #:line-color "white" #:line-
width 1)
          #:x-min -0.8 #:x-max 0.8
          #:y-min -0.8 #:y-max 0.8
          #:z-min -0.8 #:z-max 0.8
          #:altitude 25)
```



1.6 Plotting Multiple 3D Renderers

Unlike with rendering 2D plots, rendering 3D plots is order-independent. Their constituent shapes (such as polygons) are sorted by view distance and drawn back-to-front.

```
> (plot3d (list (surface3d (λ (x y) 1) #:color "LavenderBlush")
                (surface3d (λ (x y) -1) #:color "LightSteelBlue"))
          #:x-min -1 #:x-max 1 #:y-min -1 #:y-max 1)
```



Here, the top surface is first in the list, but the bottom surface is drawn first.

1.7 Plotting to Files

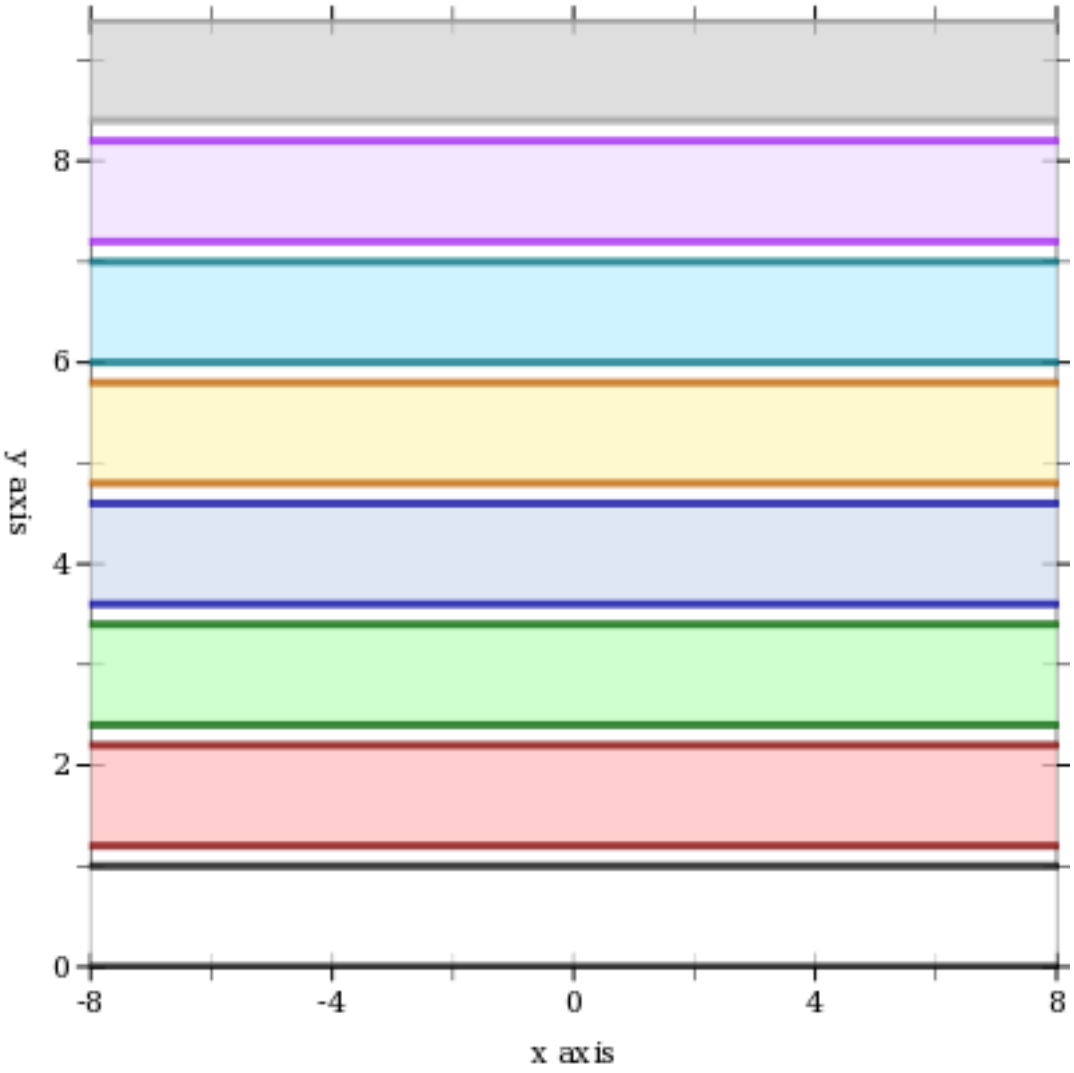
Any plot can be rendered to PNG, PDF, PS and SVG files using `plot->file` and `plot3d->file`, to include in papers and other published media.

1.8 Colors and Styles

In papers, stick to dark, fully saturated colors for lines, and light, desaturated colors for areas and surfaces. Papers are often printed in black and white, and sticking to this guideline will help black-and-white versions of color plots turn out nicely.

To make this easy, PLoT provides numbered colors that follow these guidelines, that are designed for high contrast in color as well. When used as line colors, numbers are interpreted as dark, fully saturated colors. When used as area or surface colors, numbers are interpreted as light, desaturated colors.

```
> (parameterize ([interval-line1-width  3]
                 [interval-line2-width  3])
    (plot (for/list ([i  (in-range 8)])
            (function-interval
             (λ (x) (* i 1.2)) (λ (x) (+ 1 (* i 1.2)))
             #:color i #:line1-color i #:line2-color i))
          #:x-min -8 #:x-max 8))
```

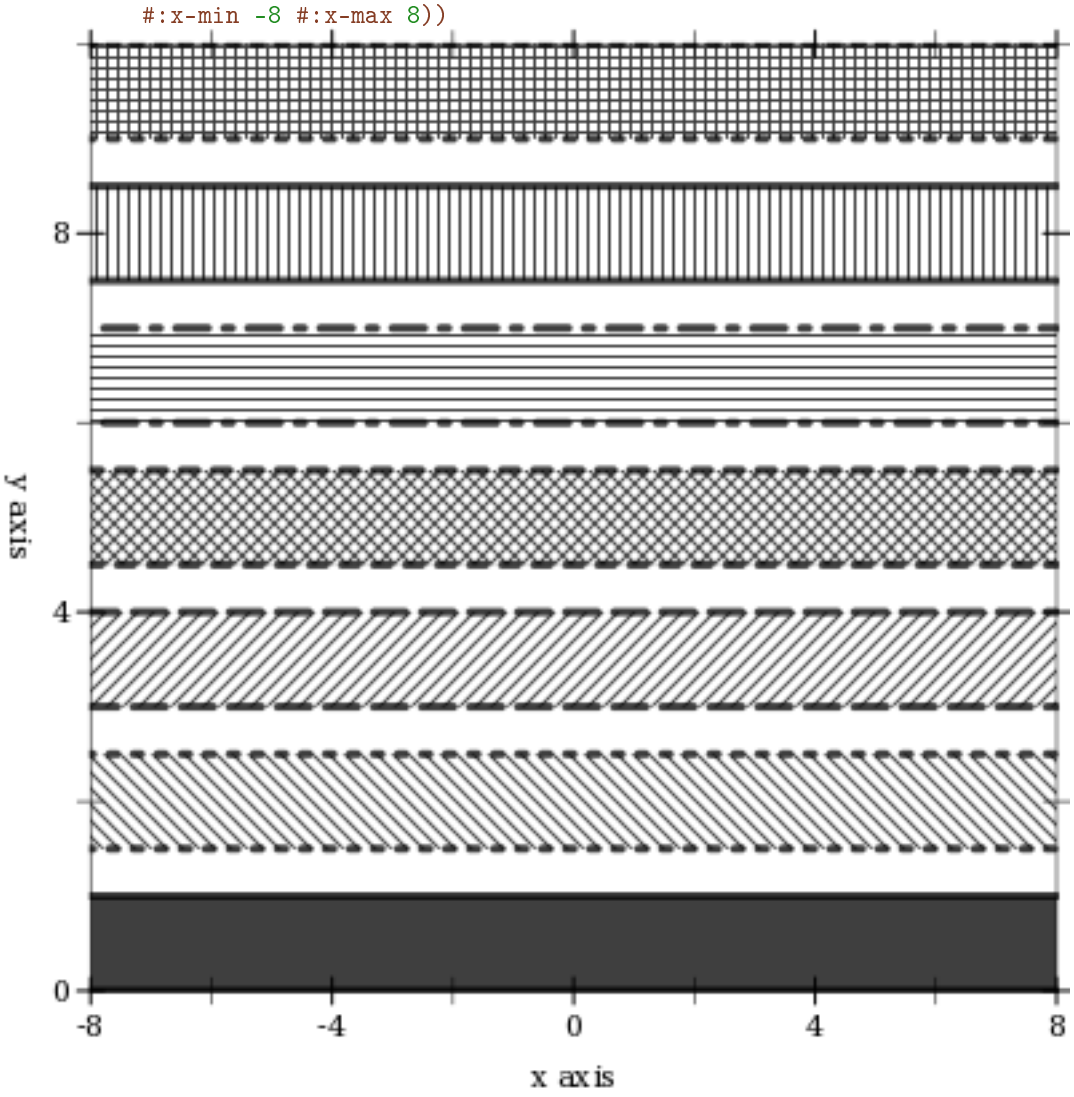


The colors repeat after 7; i.e. colors 8..15 are identical to colors 0..7.

If the paper will be published in black and white, use styles as well. There are 5 numbered pen styles and 7 numbered brush styles, which also repeat.

```
> (parameterize ([line-color  "black"]
                 [interval-color  "black"]
                 [interval-line1-color  "black"]
                 [interval-line2-color  "black"]
                 [interval-line1-width  3]
                 [interval-line2-width  3])
```

```
(plot (for/list ([i  (in-range 7)])
        (function-interval
         (λ (x) (* i 1.5)) (λ (x) (+ 1 (* i 1.5)))
         #:style i #:line1-style i #:line2-style i))
      #:x-min -8 #:x-max 8))
```



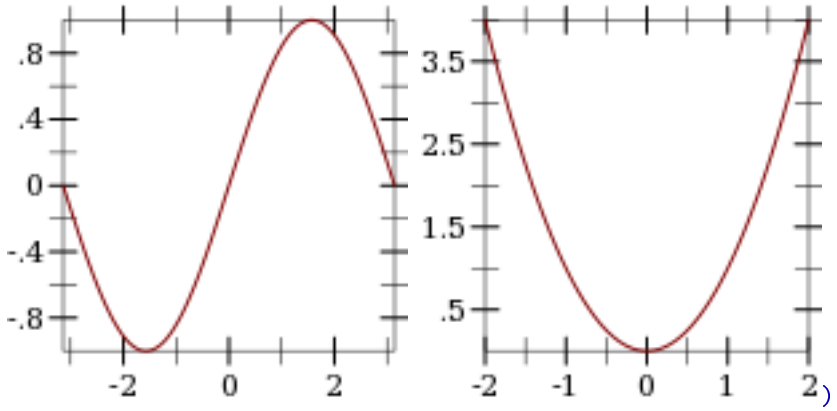
2 2D Plot Procedures

```
(plot renderer-tree
      [#:x-min x-min
       #:x-max x-max
       #:y-min y-min
       #:y-max y-max
       #:width width
       #:height height
       #:title title
       #:x-label x-label
       #:y-label y-label
       #:legend-anchor legend-anchor
       #:out-file out-file
       #:out-kind out-kind])
 → (or/c (is-a?/c image-snip%) void?)
  renderer-tree : (treeof renderer2d?)
  x-min : (or/c real? #f) = #f
  x-max : (or/c real? #f) = #f
  y-min : (or/c real? #f) = #f
  y-max : (or/c real? #f) = #f
  width : exact-positive-integer? = (plot-width)
  height : exact-positive-integer? = (plot-height)
  title : (or/c string? #f) = (plot-title)
  x-label : (or/c string? #f) = (plot-x-label)
  y-label : (or/c string? #f) = (plot-y-label)
  legend-anchor : anchor/c = (plot-legend-anchor)
  out-file : (or/c path-string? output-port? #f) = #f
  out-kind : (one-of/c 'auto 'png 'jpeg 'xmb 'xpm 'bmp 'ps 'pdf 'svg)
           = 'auto
```

Plots a 2D renderer or list of renderers (or more generally, a tree of renderers), as returned by `points`, `function`, `contours`, `discrete-histogram`, and others.

By default, `plot` produces a Racket value that is displayed as an image and can be manipulated like any other value. For example, they may be put in lists:

```
> (parameterize ([plot-width    150]
                 [plot-height   150]
                 [plot-x-label  #f]
                 [plot-y-label  #f])
    (list (plot (function sin (- pi) pi))
          (plot (function sqr -2 2))))
```

(list 

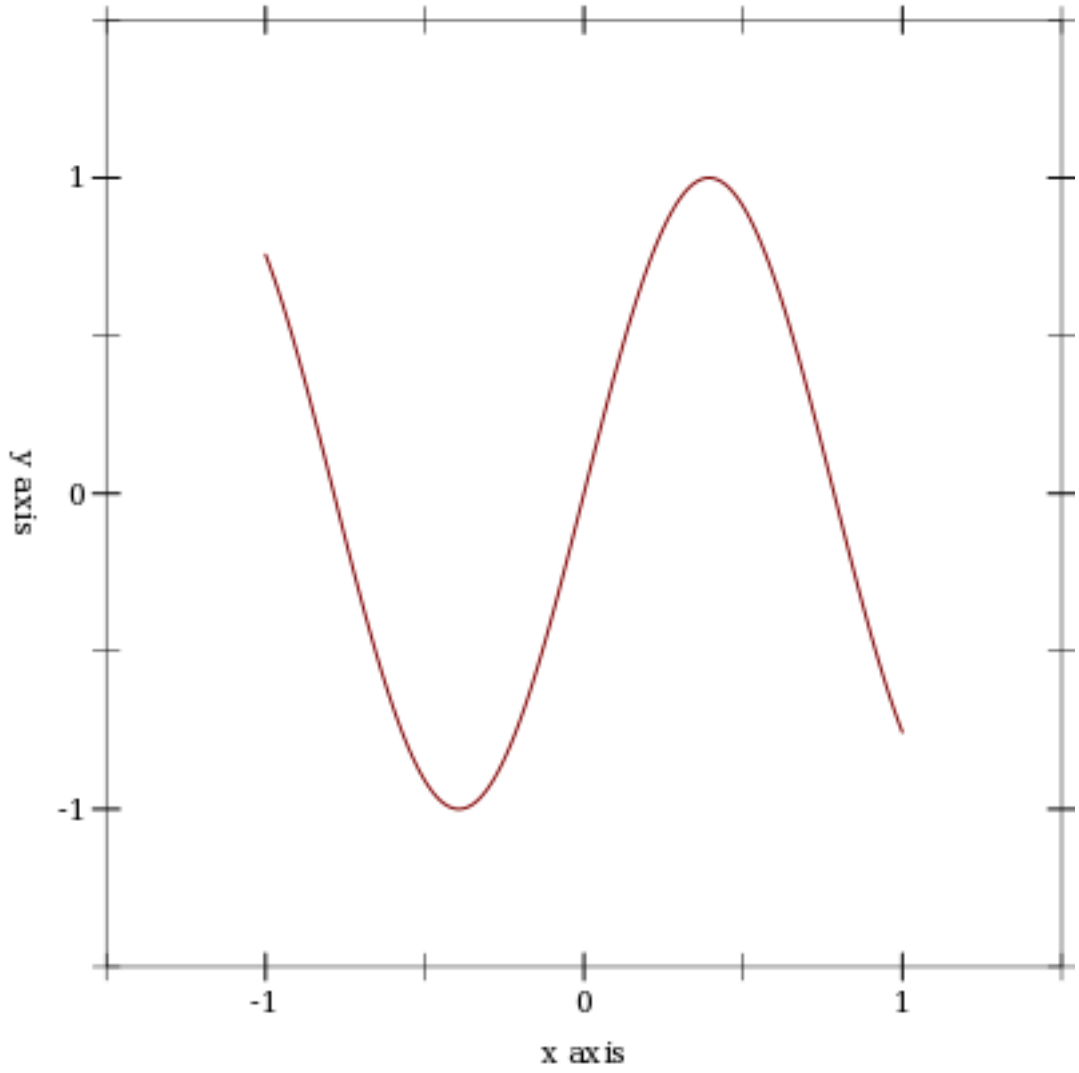
)

When the parameter `plot-new-window?` is `#t`, `plot` opens a new window to display the plot and returns `(void)`.

When `#:out-file` is given, `plot` writes the plot to a file using `plot-file` as well as returning an `image-snip%` or opening a new window.

When given, the *x-min*, *x-max*, *y-min* and *y-max* arguments determine the bounds of the plot, but not the bounds of the renderers. For example,

```
> (plot (function (λ (x) (sin (* 4 x))) -1 1)
        #:x-min -1.5 #:x-max 1.5 #:y-min -1.5 #:y-max 1.5)
```



Here, the renderer draws in [-1,1] × [-1,1], but the plot area is [-1.5,1.5] × [-1.5,1.5].

Deprecated keywords. The `#:fgcolor` and `#:bgcolor` keyword arguments are currently supported for backward compatibility, but may not be in the future. Please set the `plot-foreground` and `plot-background` parameters instead of using these keyword arguments. The `#:lncolor` keyword argument is also accepted for backward compatibility but deprecated. It does nothing.

```
(plot-file  renderer-tree
            output
           [kind]
            #:<plot-keyword> <plot-keyword> ...) → void?
  renderer-tree : (treeof renderer2d?)
  output : (or/c path-string? output-port?)
  kind : (one-of/c 'auto 'png 'jpeg 'xmb 'xpm 'bmp 'ps 'pdf 'svg)
       = 'auto
  <plot-keyword> : <plot-keyword-contract>
(plot-pict renderer-tree ...) → pict?
  renderer-tree : (treeof renderer2d?)
(plot-bitmap renderer-tree ...) → (is-a?/c bitmap%)
  renderer-tree : (treeof renderer2d?)
(plot-snip renderer-tree ...) → (is-a?/c image-snip%)
  renderer-tree : (treeof renderer2d?)
(plot-frame renderer-tree ...) → (is-a?/c frame%)
  renderer-tree : (treeof renderer2d?)
```

Plot to different backends. Each of these procedures has the same keyword arguments as `plot`, except for deprecated keywords.

Use `plot-file` to save a plot to a file. When creating a JPEG file, the parameter `plot-jpeg-quality` determines its quality. When creating a PostScript or PDF file, the parameter `plot-ps/pdf-interactive?` determines whether the user is given a dialog for setting printing parameters. (See `post-script-dc%` and `pdf-dc%`.) When *kind* is `'auto`, `plot-file` tries to determine the kind of file to write from the file name extension.

Use `plot-pict` to plot to a slideshow `pict`. For example,

```
#lang slideshow
(require plot)

(plot-font-size (current-font-size))
(plot-width (current-para-width))
(plot-height 600)
(plot-background-alpha 1/2)

(slide
 #:title "A 2D Parabola"
 (plot-pict (function sqr -1 1 #:label "y = x^2")))
```

creates a slide containing a 2D plot of a parabola.

Use `plot-bitmap` to create a `bitmap%`.

Use plot-frame to create a frame% regardless of the value of plot-new-window?. The frame is initially hidden.

Use plot-snip to create an image-snip% regardless of the value of plot-new-window?.

```
(plot/dc  renderer-tree
          dc
          x
          y
          width
          height
          [#:x-min x-min
          #:x-max x-max
          #:y-min y-min
          #:y-max y-max
          #:title title
          #:x-label x-label
          #:y-label y-label
          #:legend-anchor legend-anchor]) → void?
  renderer-tree : (treeof renderer2d?)
  dc : (is-a?/c dc<%>)
  x : real?
  y : real?
  width : (>=/c 0)
  height : (>=/c 0)
  x-min : (or/c real? #f) = #f
  x-max : (or/c real? #f) = #f
  y-min : (or/c real? #f) = #f
  y-max : (or/c real? #f) = #f
  title : (or/c string? #f) = (plot-title)
  x-label : (or/c string? #f) = (plot-x-label)
  y-label : (or/c string? #f) = (plot-y-label)
  legend-anchor : anchor/c = (plot-legend-anchor)
```

Plots to an arbitrary device context, in the rectangle with width *width*, height *height*, and upper-left corner *x*,*y*.

Every §2 “2D Plot Procedures” procedure is defined in terms of plot/dc.

Use this if you need to continually update a plot on a canvas%, or to create other plot-like functions with different backends.

3 2D Renderers

```
(renderer2d? value) → boolean?
  value : any/c
```

Returns `#t` if *value* is a 2D renderer; that is, if `plot` can plot *value*. The following functions create such renderers.

3.1 2D Renderer Function Arguments

Functions that return 2D renderers always have these kinds of arguments:

- Required (and possibly optional) arguments representing the graph to plot.
- Optional keyword arguments for overriding calculated bounds, with the default value `#f`.
- Optional keyword arguments that determine the appearance of the plot.
- The optional keyword argument `#:label`, which specifies the name of the renderer in the legend.

We will take `function`, perhaps the most commonly used renderer-producing function, as an example.

Graph arguments. The first argument to `function` is the required `f`, the function to plot. It is followed by two optional arguments `x-min` and `x-max`, which specify the renderer's x bounds. (If not given, the x bounds will be the plot area x bounds, as requested by another renderer or specified to `plot` using `#:x-min` and `#:x-max`.)

These three arguments define the *graph* of the function `f`, a possibly infinite set of pairs of points `x`,`(f x)`. An infinite graph cannot be plotted directly, so the renderer must approximately plot the points in it. The renderer returned by `function` does this by drawing lines connected end-to-end.

Overriding bounds arguments. Next in `function`'s argument list are the keyword arguments `#:y-min` and `#:y-max`, which override the renderer's calculated y bounds if given.

Appearance arguments. The next keyword argument is `#:samples`, which determines the quality of the renderer's approximate plot (higher is better). Following `#:samples` are `#:color`, `#:width`, `#:style` and `#:alpha`, which determine the color, width, style and opacity of the lines comprising the plot.

In general, the keyword arguments that determine the appearance of plots follow consistent naming conventions. The most common keywords are `#:color` (for fill and line colors),

`#:width` (for line widths), `#:style` (for fill and line styles) and `#:alpha`. When a function needs both a fill color and a line color, the fill color is given using `#:color`, and the line color is given using `#:line-color` (or some variation, such as `#:line1-color`). Styles follow the same rule.

Every appearance keyword argument defaults to the value of a parameter. This allows whole families of plots to be altered with little work. For example, setting `(line-color 3)` causes every subsequent renderer that draws connected lines to draw its lines in blue.

Label argument. Lastly, there is `#:label`. If given, the `function` renderer will generate a label entry that `plot` puts in the legend.

Not every renderer-producing function has a `#:label` argument; for example, `error-bars`.

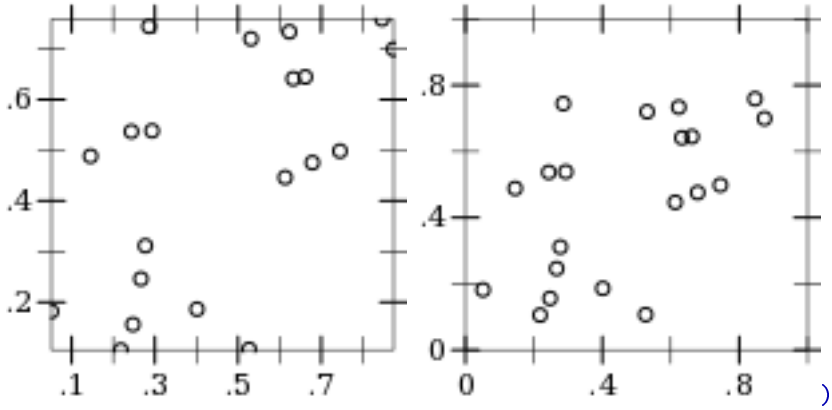
3.2 2D Point Renderers

```
(points vs
        [#:x-min x-min
         #:x-max x-max
         #:y-min y-min
         #:y-max y-max
         #:sym sym
         #:color color
         #:size size
         #:line-width line-width
         #:alpha alpha
         #:label label])          → renderer2d?
  vs : (listof (vector/c real? real?))
  x-min : (or/c real? #f) = #f
  x-max : (or/c real? #f) = #f
  y-min : (or/c real? #f) = #f
  y-max : (or/c real? #f) = #f
  sym : point-sym/c = (point-sym)
  color : plot-color/c = (point-color)
  size : (>=/c 0) = (point-size)
  line-width : (>=/c 0) = (point-line-width)
  alpha : (real-in 0 1) = (point-alpha)
  label : (or/c string? #f) = #f
```

Returns a renderer that draws points. Use it, for example, to draw 2D scatter plots.

The renderer sets its bounds to the smallest rectangle that contains the points. Still, it is often necessary to override these bounds, especially with randomized data. For example,

```
> (parameterize ([plot-width    150]
                 [plot-height   150]
                 [plot-x-label  #f]
                 [plot-y-label  #f])
    (define xs (build-list 20 (λ _ (random))))
    (define ys (build-list 20 (λ _ (random))))
    (list (plot (points (map vector xs ys)))
          (plot (points (map vector xs ys)
                        #:x-min 0 #:x-max 1
                        #:y-min 0 #:y-max 1))))
```




```
(list                                                    )
```

Readers of the first plot could only guess that the random points were generated in [0,1] × [0,1].

The `#:sym` argument may be any integer, a Unicode character or string, or a symbol in `known-point-symbols`. Use an integer when you need different points but don't care exactly what they are.

```
(vector-field  f
               [x-min
                x-max
                y-min
                y-max
                #:samples samples
                #:scale scale
                #:color color
                #:line-width line-width
                #:line-style line-style
                #:alpha alpha
                #:label label])          → renderer2d?
  f : (or/c (real? real? . -> . (vector/c real? real?))
            ((vector/c real? real?) . -> . (vector/c real? real?)))
  x-min : (or/c real? #f) = #f
```

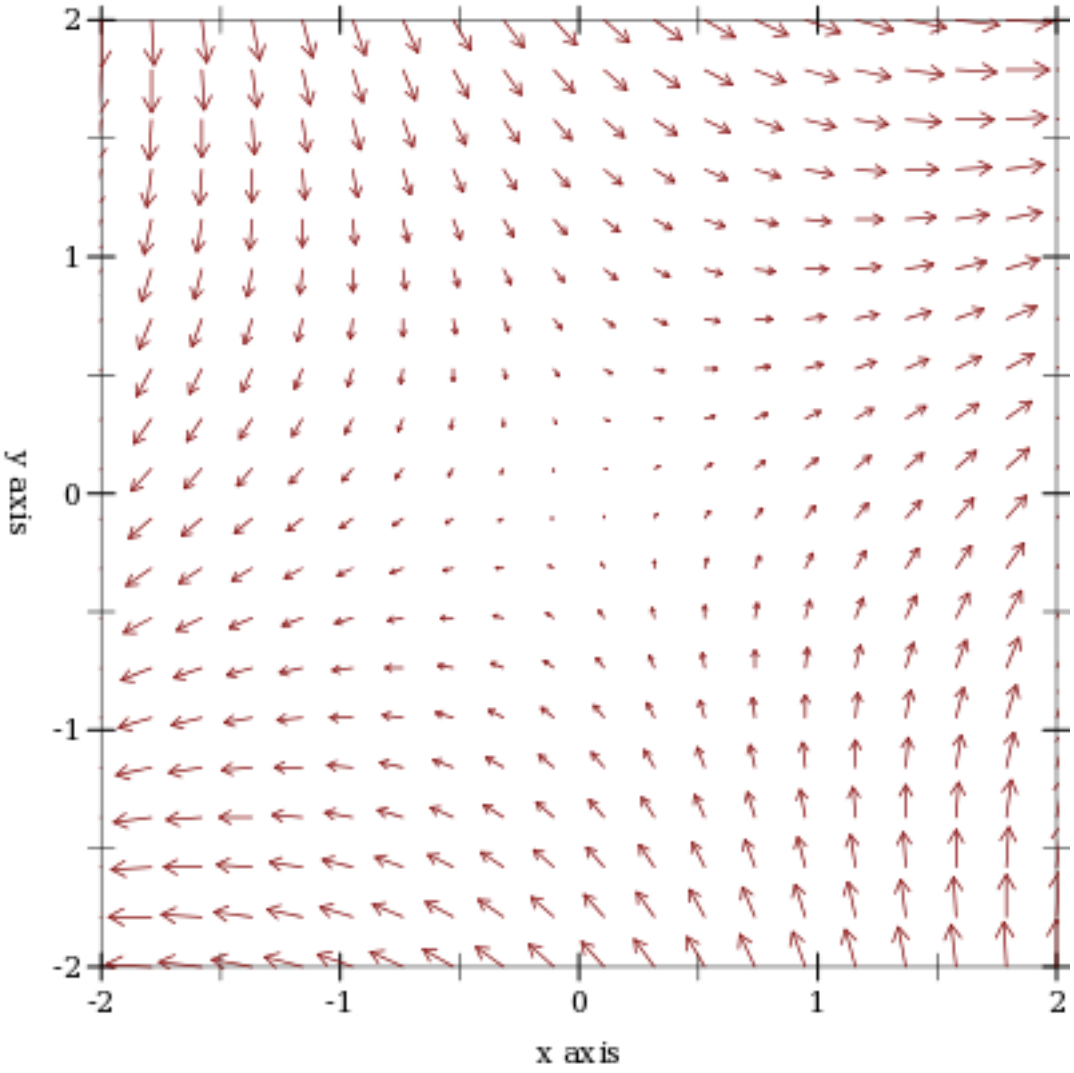
```
x-max : (or/c real? #f) = #f
y-min : (or/c real? #f) = #f
y-max : (or/c real? #f) = #f
samples : exact-positive-integer? = (vector-field-samples)
scale : (or/c real? (one-of/c 'auto 'normalized))
      = (vector-field-scale)
color : plot-color/c = (vector-field-color)
line-width : (>=/c 0) = (vector-field-line-width)
line-style : plot-pen-style/c = (vector-field-line-style)
alpha : (real-in 0 1) = (vector-field-alpha)
label : (or/c string? #f) = #f
```

Returns a renderer that draws a vector field.

If *scale* is a real number, arrow lengths are multiplied by *scale*. If 'auto, the scale is calculated in a way that keeps arrows from overlapping. If 'normalized, each arrow is made the same length: the maximum length that would have been allowed by 'auto.

An example of automatic scaling:

```
> (plot (vector-field (λ (x y) (vector (+ x y) (- x y)))
                      -2 2 -2 2))
```

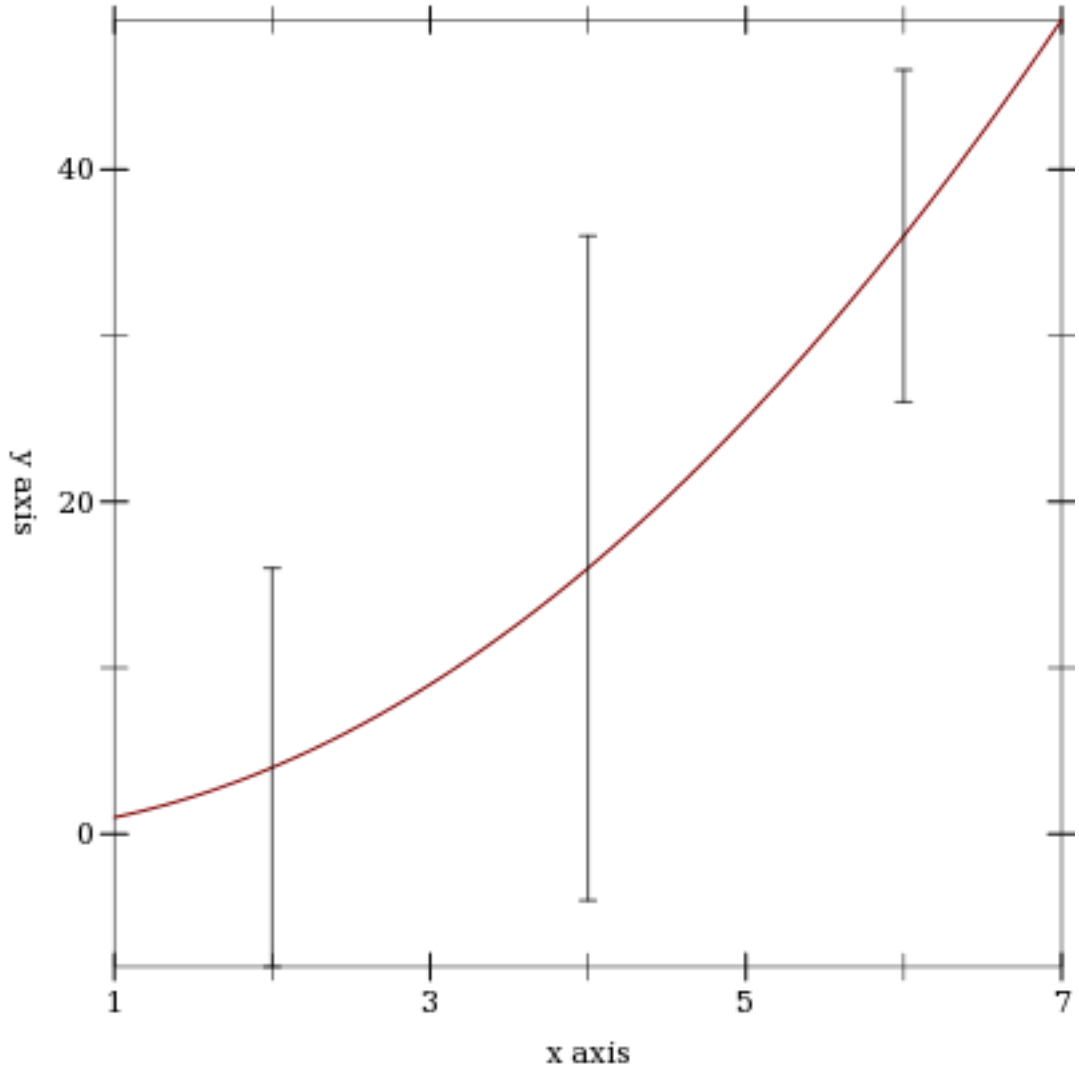



```
(error-bars  bars
             [#:x-min x-min
              #:x-max x-max
              #:y-min y-min
              #:y-max y-max
              #:color color
              #:line-width line-width
              #:line-style line-style
              #:width width
              #:alpha alpha])          → renderer2d?
  bars : (listof (vector/c real? real? real?))
```

```
x-min : (or/c real? #f) = #f
x-max : (or/c real? #f) = #f
y-min : (or/c real? #f) = #f
y-max : (or/c real? #f) = #f
color : plot-color/c = (error-bar-color)
line-width : (>=/c 0) = (error-bar-line-width)
line-style : plot-pen-style/c = (error-bar-line-style)
width : (>=/c 0) = (error-bar-width)
alpha : (real-in 0 1) = (error-bar-alpha)
```

Returns a renderer that draws error bars. The first and second element in each vector in *bars* comprise the coordinate; the third is the height.

```
> (plot (list (function sqr 1 7)
              (error-bars (list (vector 2 4 12)
                                (vector 4 16 20)
                                (vector 6 36 10)))))
```

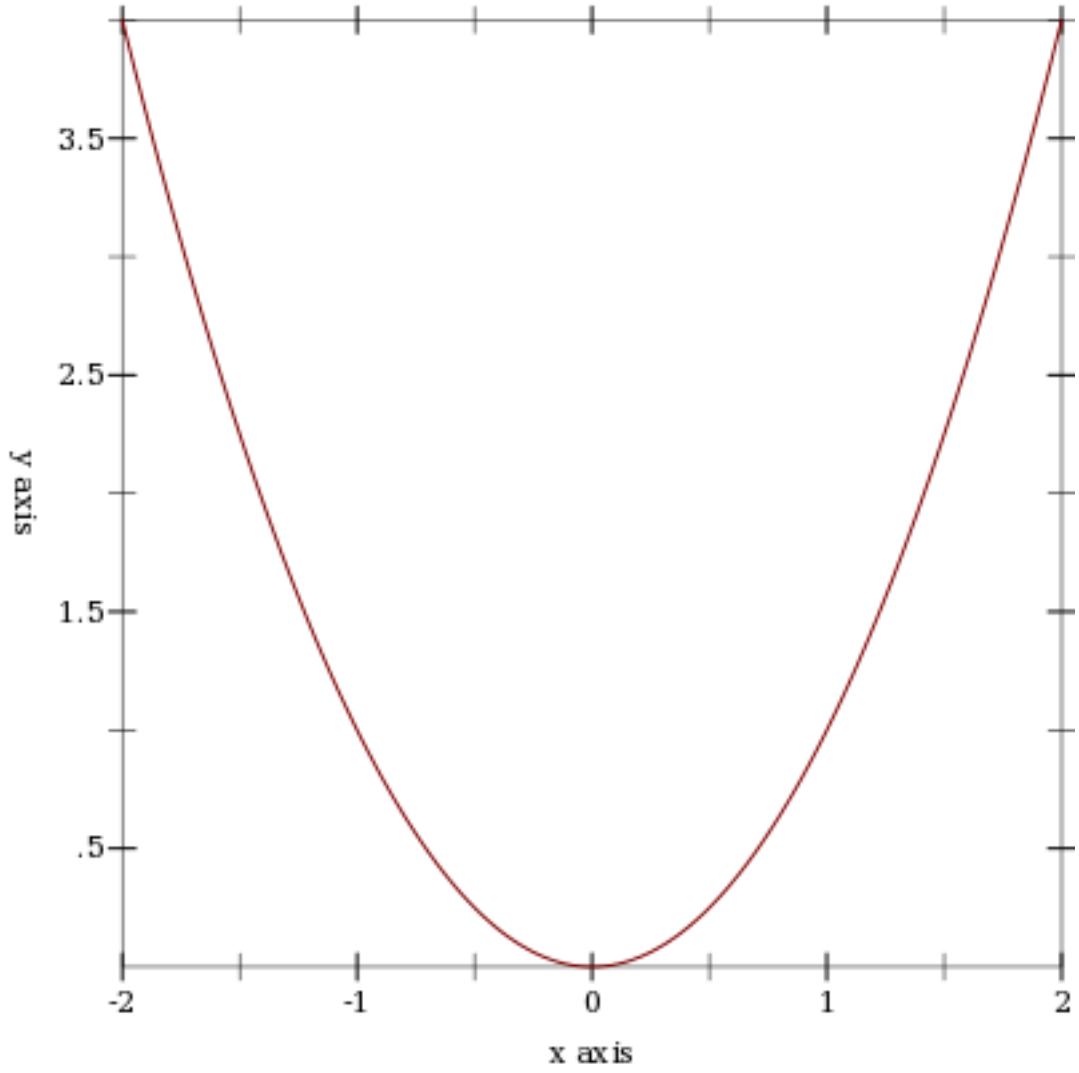


3.3 2D Line Renderers

```
(function  f
           [x-min
            x-max
            #:y-min y-min
            #:y-max y-max
            #:samples samples
            #:color color
            #:width width
            #:style style
            #:alpha alpha
            #:label label])    → renderer2d?
  f : (real? . -> . real?)
  x-min : (or/c real? #f) = #f
  x-max : (or/c real? #f) = #f
  y-min : (or/c real? #f) = #f
  y-max : (or/c real? #f) = #f
  samples : (and/c exact-integer? (>=/c 2)) = (line-samples)
  color : plot-color/c = (line-color)
  width : (>=/c 0) = (line-width)
  style : plot-pen-style/c = (line-style)
  alpha : (real-in 0 1) = (line-alpha)
  label : (or/c string? #f) = #f
```

Returns a renderer that plots a function of x. For example, a parabola:

```
> (plot (function sqr -2 2))
```

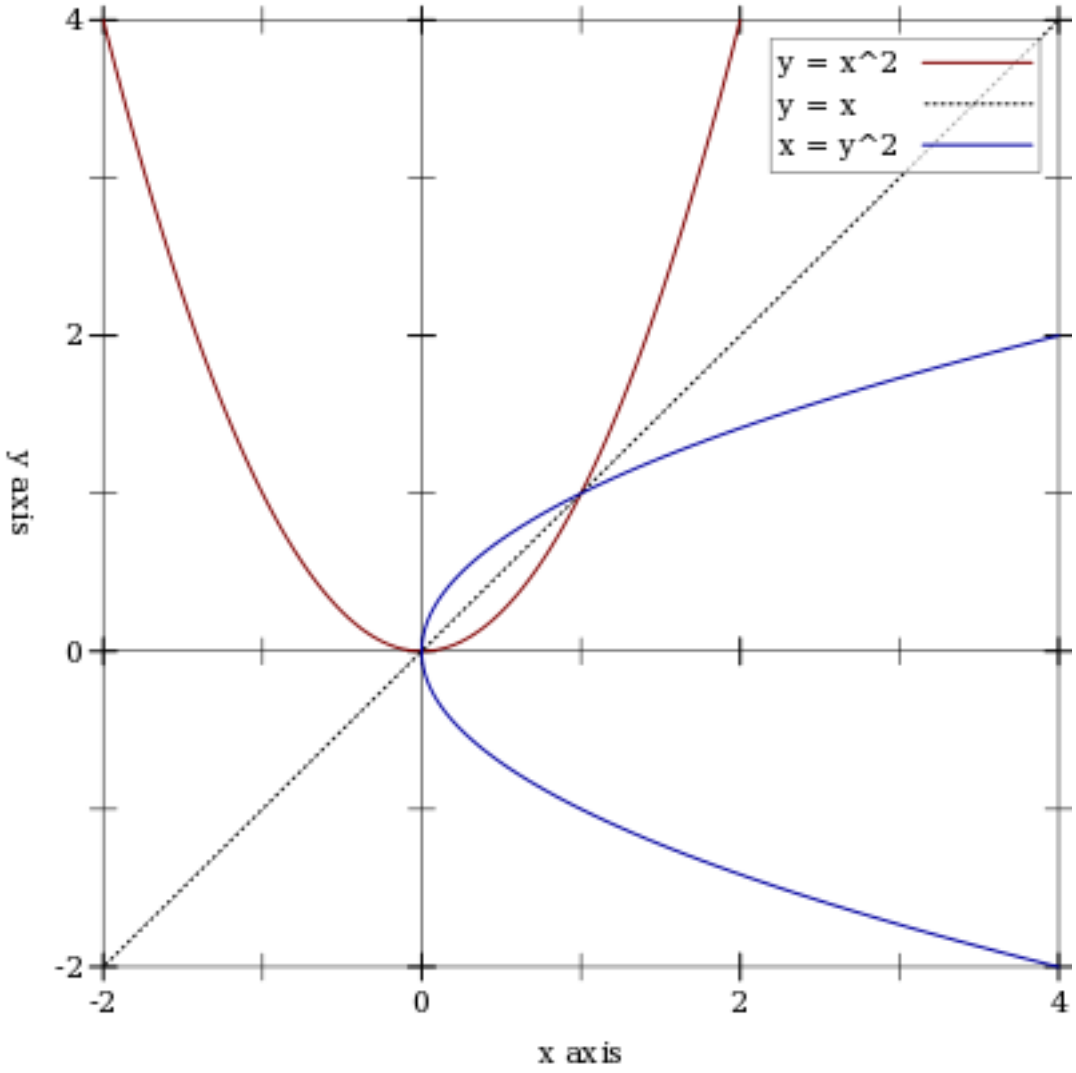



```
(inverse  f
          [y-min
           y-max
           #:x-min x-min
           #:x-max x-max
           #:samples samples
           #:color color
           #:width width
           #:style style
           #:alpha alpha
           #:label label])    → renderer2d?
```

```
f : (real? . -> . real?)
y-min : (or/c real? #f) = #f
y-max : (or/c real? #f) = #f
x-min : (or/c real? #f) = #f
x-max : (or/c real? #f) = #f
samples : (and/c exact-integer? (>=/c 2)) = (line-samples)
color : plot-color/c = (line-color)
width : (>=/c 0) = (line-width)
style : plot-pen-style/c = (line-style)
alpha : (real-in 0 1) = (line-alpha)
label : (or/c string? #f) = #f
```

Like `function`, but regards *f* as a function of *y*. For example, a parabola, an inverse parabola, and the reflection line:

```
> (plot (list (axes)
              (function sqr -2 2 #:label "y = x^2")
              (function (λ (x) x) #:color 0 #:style 'dot #:label "y
= x")
              (inverse sqr -2 2 #:color 3 #:label "x = y^2")))
```

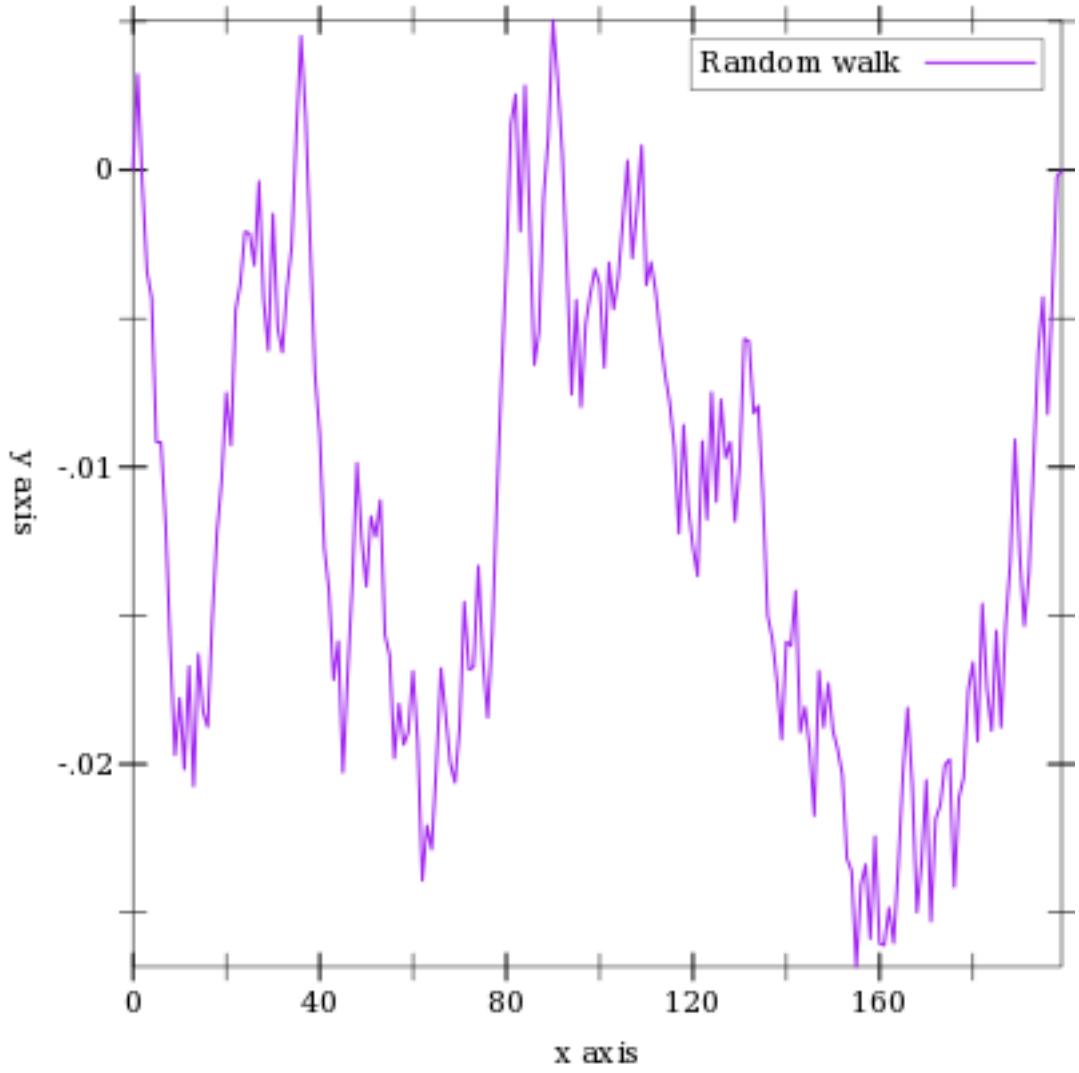



```
(lines  vs
        [#:x-min x-min
         #:x-max x-max
         #:y-min y-min
         #:y-max y-max
         #:color color
         #:width width
         #:style style
         #:alpha alpha
         #:label label]) → renderer2d?
  vs : (listof (vector/c real? real?))
```

```
  x-min : (or/c real? #f) = #f
  x-max : (or/c real? #f) = #f
  y-min : (or/c real? #f) = #f
  y-max : (or/c real? #f) = #f
  color : plot-color/c = (line-color)
  width : (>=/c 0) = (line-width)
  style : plot-pen-style/c = (line-style)
  alpha : (real-in 0 1) = (line-alpha)
  label : (or/c string? #f) = #f
```

Returns a renderer that draws lines. This is directly useful for plotting a time series, such as a random walk:

```
> (plot (lines
         (reverse
          (for/fold ([lst (list (vector 0 0))]) ([i  (in-
range 1 200)])
            (match-define (vector x y) (first lst))
            (cons (vector i (+ y (* 1/100 (- (random) 1/2)))) lst)))
         #:color 6 #:label "Random walk"))
```

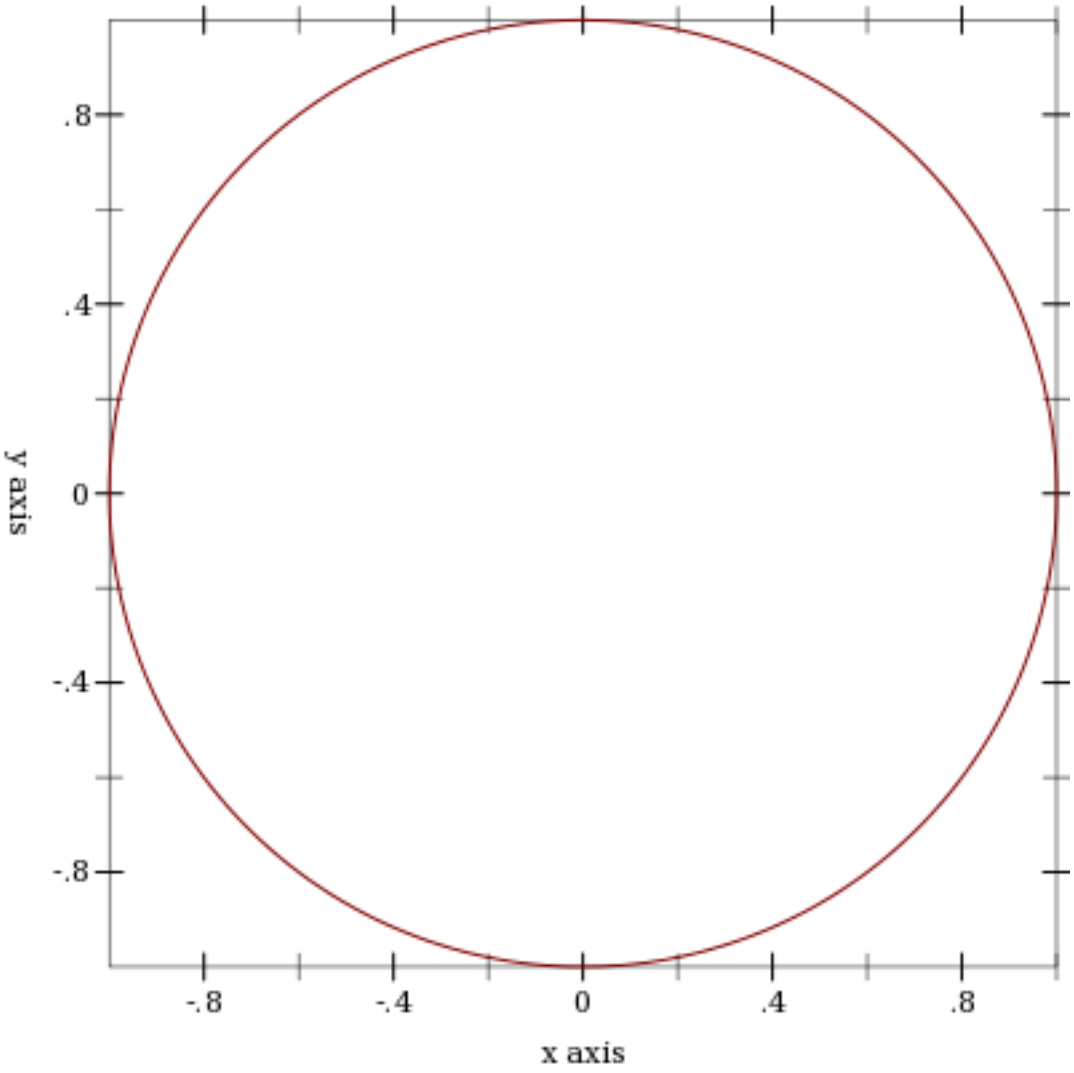


The `parametric` and `polar` functions are defined using `lines`.

```
(parametric  f
             t-min
             t-max
            [#:x-min x-min
             #:x-max x-max
             #:y-min y-min
             #:y-max y-max
             #:samples samples
             #:color color
             #:width width
             #:style style
             #:alpha alpha
             #:label label])    → renderer2d?
  f : (real? . -> . (vector/c real? real?))
  t-min : real?
  t-max : real?
  x-min : (or/c real? #f) = #f
  x-max : (or/c real? #f) = #f
  y-min : (or/c real? #f) = #f
  y-max : (or/c real? #f) = #f
  samples : (and/c exact-integer? (>=/c 2)) = (line-samples)
  color : plot-color/c = (line-color)
  width : (>=/c 0) = (line-width)
  style : plot-pen-style/c = (line-style)
  alpha : (real-in 0 1) = (line-alpha)
  label : (or/c string? #f) = #f
```

Returns a renderer that plots vector-valued functions of time. For example, the circle as a function of time can be plotted using

```
> (plot (parametric (λ (t) (vector (cos t) (sin t))) 0 (* 2 pi)))
```

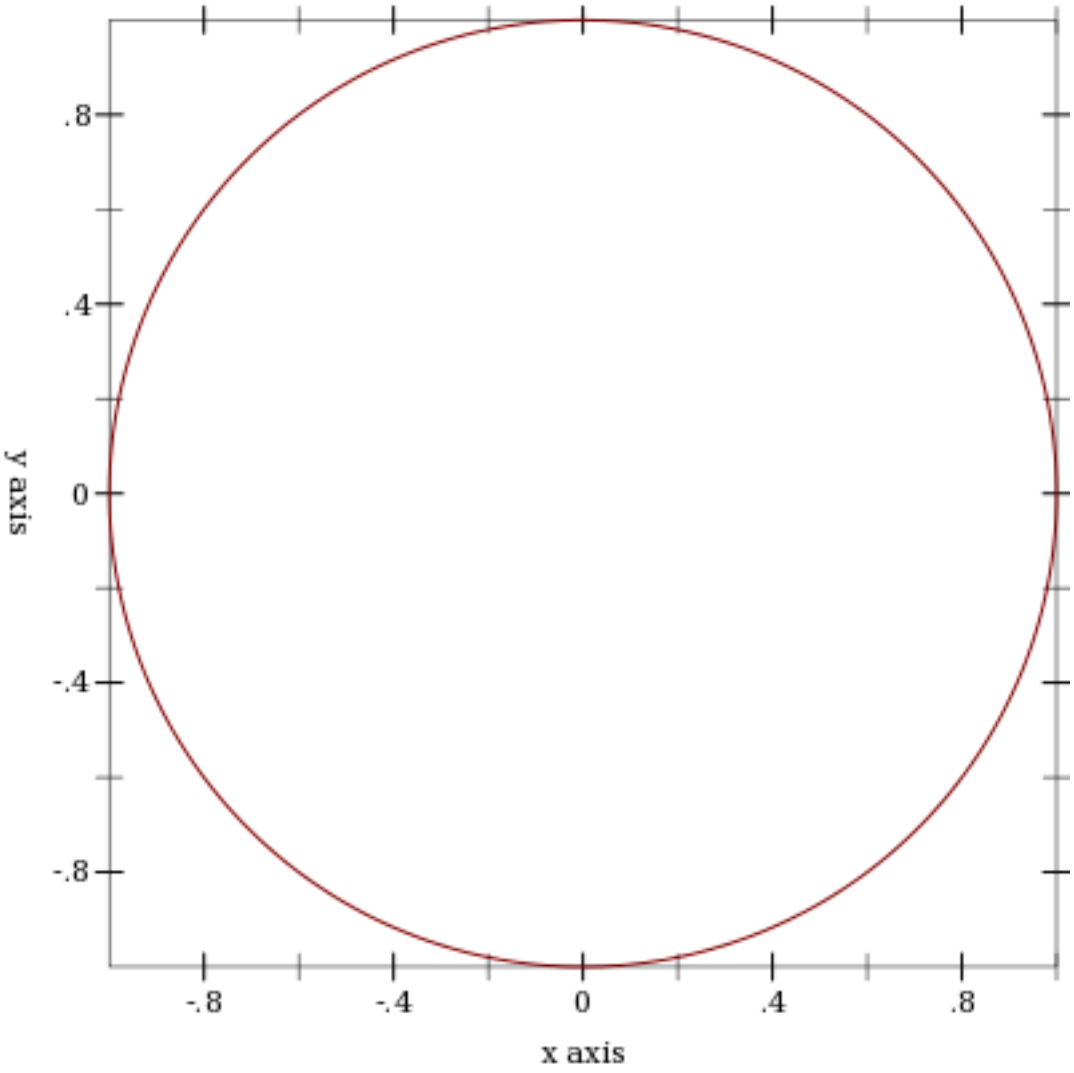

.8
.4
0
-.4
-.8
y axis
-.8
-.4
0
.4
.8
x axis

```
(polar  f
       [θ-min
        θ-max
        #:x-min x-min
        #:x-max x-max
        #:y-min y-min
        #:y-max y-max
        #:samples samples
        #:color color
        #:width width
        #:style style
        #:alpha alpha
        #:label label])   → renderer2d?
  f : (real? . -> . real?)
  θ-min : real? = 0
  θ-max : real? = (* 2 pi)
  x-min : (or/c real? #f) = #f
  x-max : (or/c real? #f) = #f
  y-min : (or/c real? #f) = #f
  y-max : (or/c real? #f) = #f
  samples : (and/c exact-integer? (>=/c 2)) = (line-samples)
  color : plot-color/c = (line-color)
  width : (>=/c 0) = (line-width)
  style : plot-pen-style/c = (line-style)
  alpha : (real-in 0 1) = (line-alpha)
  label : (or/c string? #f) = #f
```

Returns a renderer that plots functions from angle to radius. Note that the angle parameters *θ-min* and *θ-max* default to `0` and `(* 2 pi)`.

For example, drawing a full circle:

```
> (plot (polar (λ (θ) 1)))
```

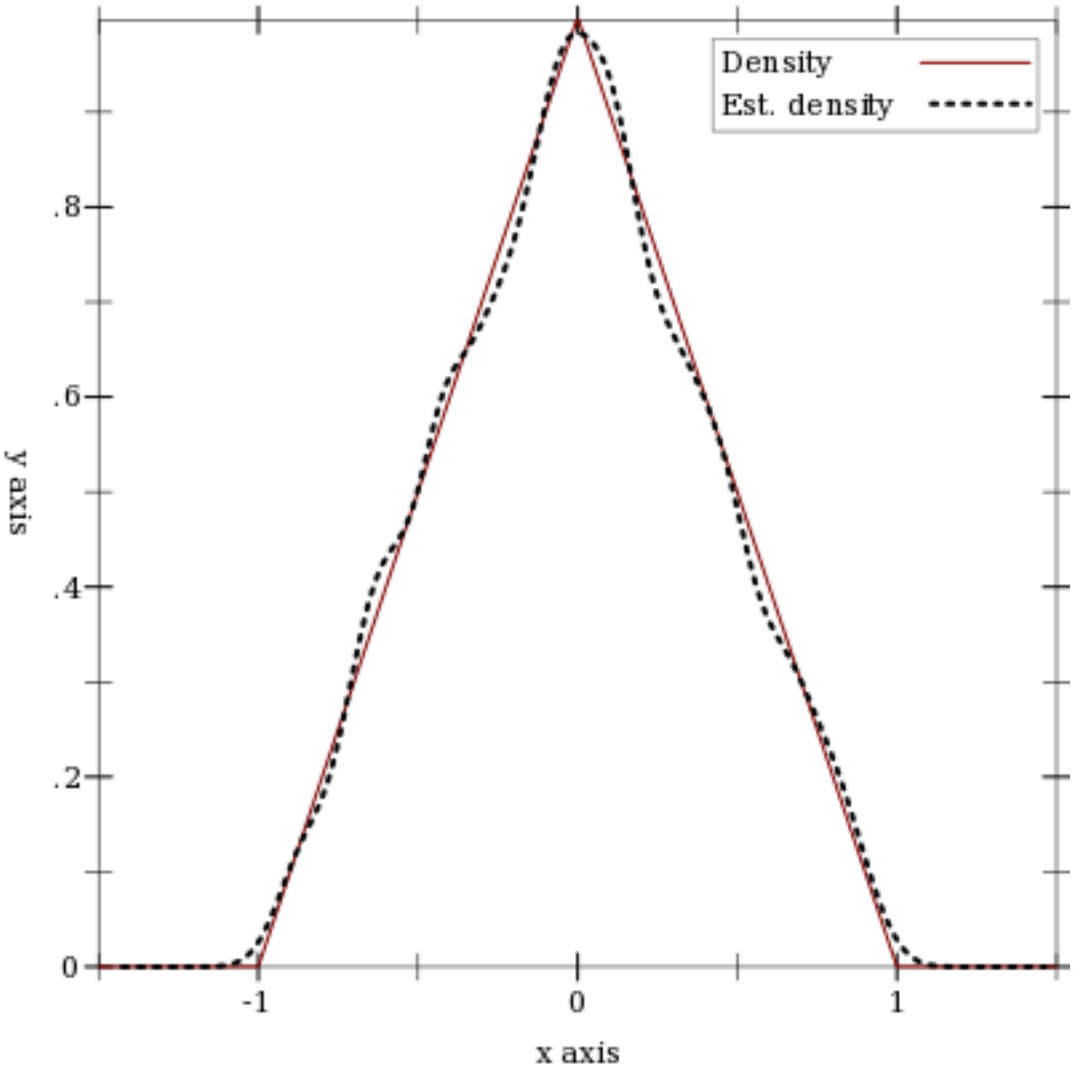

.8
.4
0
-.4
-.8
y axis
-.8
-.4
0
.4
.8
x axis

```
(density  xs
          [bw-adjust
           #:x-min x-min
           #:x-max x-max
           #:y-min y-min
           #:y-max y-max
           #:samples samples
           #:color color
           #:width width
           #:style style
           #:alpha alpha
           #:label label])    → renderer2d?
  xs : (listof real?)
  bw-adjust : real? = 1
  x-min : (or/c real? #f) = #f
  x-max : (or/c real? #f) = #f
  y-min : (or/c real? #f) = #f
  y-max : (or/c real? #f) = #f
  samples : (and/c exact-integer? (>=/c 2)) = (line-samples)
  color : plot-color/c = (line-color)
  width : (>=/c 0) = (line-width)
  style : plot-pen-style/c = (line-style)
  alpha : (real-in 0 1) = (line-alpha)
  label : (or/c string? #f) = #f
```

Returns a renderer that plots an estimated density for the given points. The bandwidth for the kernel is calculated as `(* bw-adjust 1.06 sd (expt n -0.2))`, where `sd` is the standard deviation of the data and `n` is the number of points. Currently, the only supported kernel is the Gaussian.

For example, to plot an estimated density of the triangle distribution:

```
> (plot (list (function (λ (x) (cond [(or (x . < . -1) (x . >
. 1))  0]
                                     [(x . < . 0)   (+ 1 x)]
                                     [(x . >= . 0)  (- 1 x)]))
                        -1.5 1.5 #:label "Density")
              (density (build-list
                        2000 (λ (n) (- (+ (random) (random)) 1)))
                       #:color 0 #:width 2 #:style 'dot
                       #:label "Est. density")))
```



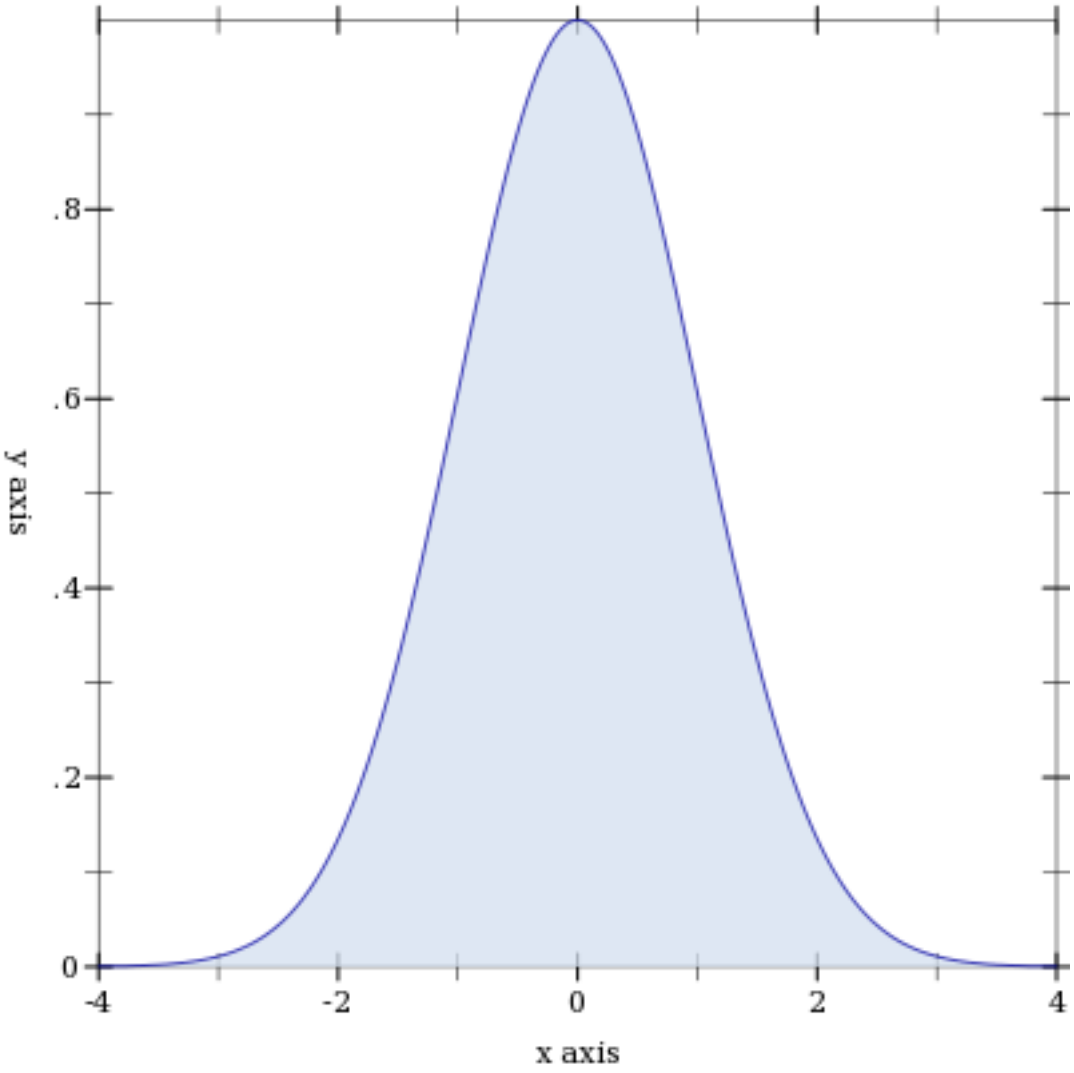
3.4 2D Interval Renderers

These renderers each correspond with a line renderer, and graph the area between two lines.

```
(function-interval  f1
                    f2
                   [x-min
                    x-max
                    #:y-min y-min
                    #:y-max y-max
                    #:samples samples
                    #:color color
                    #:style style
                    #:line1-color line1-color
                    #:line1-width line1-width
                    #:line1-style line1-style
                    #:line2-color line2-color
                    #:line2-width line2-width
                    #:line2-style line2-style
                    #:alpha alpha
                    #:label label])           → renderer2d?
  f1 : (real? . -> . real?)
  f2 : (real? . -> . real?)
  x-min : (or/c real? #f) = #f
  x-max : (or/c real? #f) = #f
  y-min : (or/c real? #f) = #f
  y-max : (or/c real? #f) = #f
  samples : (and/c exact-integer? (>=/c 2)) = (line-samples)
  color : plot-color/c = (interval-color)
  style : plot-brush-style/c = (interval-style)
  line1-color : plot-color/c = (interval-line1-color)
  line1-width : (>=/c 0) = (interval-line1-width)
  line1-style : plot-pen-style/c = (interval-line1-style)
  line2-color : plot-color/c = (interval-line2-color)
  line2-width : (>=/c 0) = (interval-line2-width)
  line2-style : plot-pen-style/c = (interval-line2-style)
  alpha : (real-in 0 1) = (interval-alpha)
  label : (or/c string? #f) = #f
```

Corresponds with `function`.

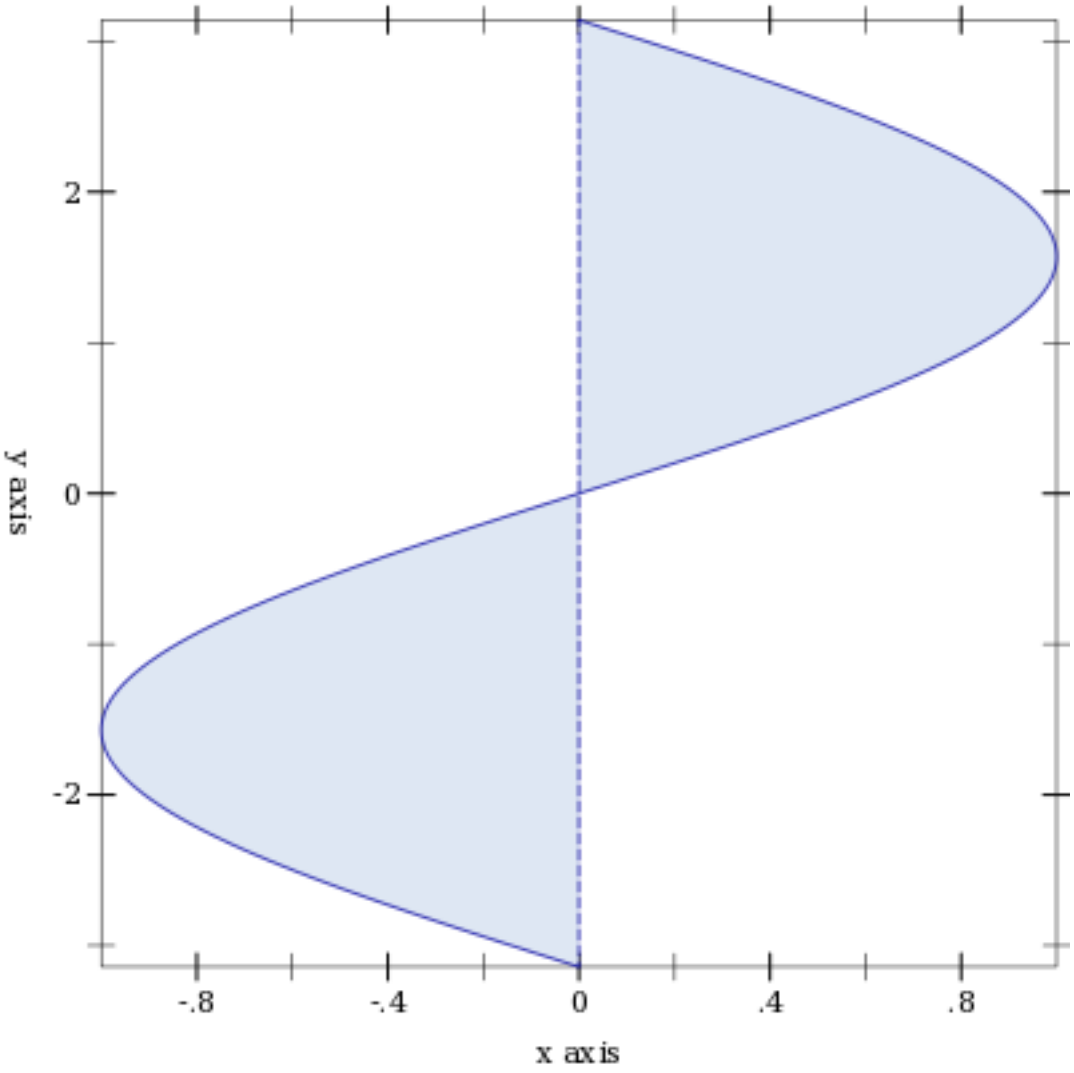
```
> (plot (function-interval (λ (x) 0) (λ (x) (exp (* -1/2 (sqr x))))
                           -4 4 #:line1-style 'transparent))
```


.8
.6
.4
.2
0
y axis
-4
-2
0
2
4
x axis

```
(inverse-interval  f1
                   f2
                  [y-min
                   y-max
                   #:x-min x-min
                   #:x-max x-max
                   #:samples samples
                   #:color color
                   #:style style
                   #:line1-color line1-color
                   #:line1-width line1-width
                   #:line1-style line1-style
                   #:line2-color line2-color
                   #:line2-width line2-width
                   #:line2-style line2-style
                   #:alpha alpha
                   #:label label])           → renderer2d?
  f1 : (real? . -> . real?)
  f2 : (real? . -> . real?)
  y-min : (or/c real? #f) = #f
  y-max : (or/c real? #f) = #f
  x-min : (or/c real? #f) = #f
  x-max : (or/c real? #f) = #f
  samples : (and/c exact-integer? (>=/c 2)) = (line-samples)
  color : plot-color/c = (interval-color)
  style : plot-brush-style/c = (interval-style)
  line1-color : plot-color/c = (interval-line1-color)
  line1-width : (>=/c 0) = (interval-line1-width)
  line1-style : plot-pen-style/c = (interval-line1-style)
  line2-color : plot-color/c = (interval-line2-color)
  line2-width : (>=/c 0) = (interval-line2-width)
  line2-style : plot-pen-style/c = (interval-line2-style)
  alpha : (real-in 0 1) = (interval-alpha)
  label : (or/c string? #f) = #f
```

Corresponds with `inverse`.

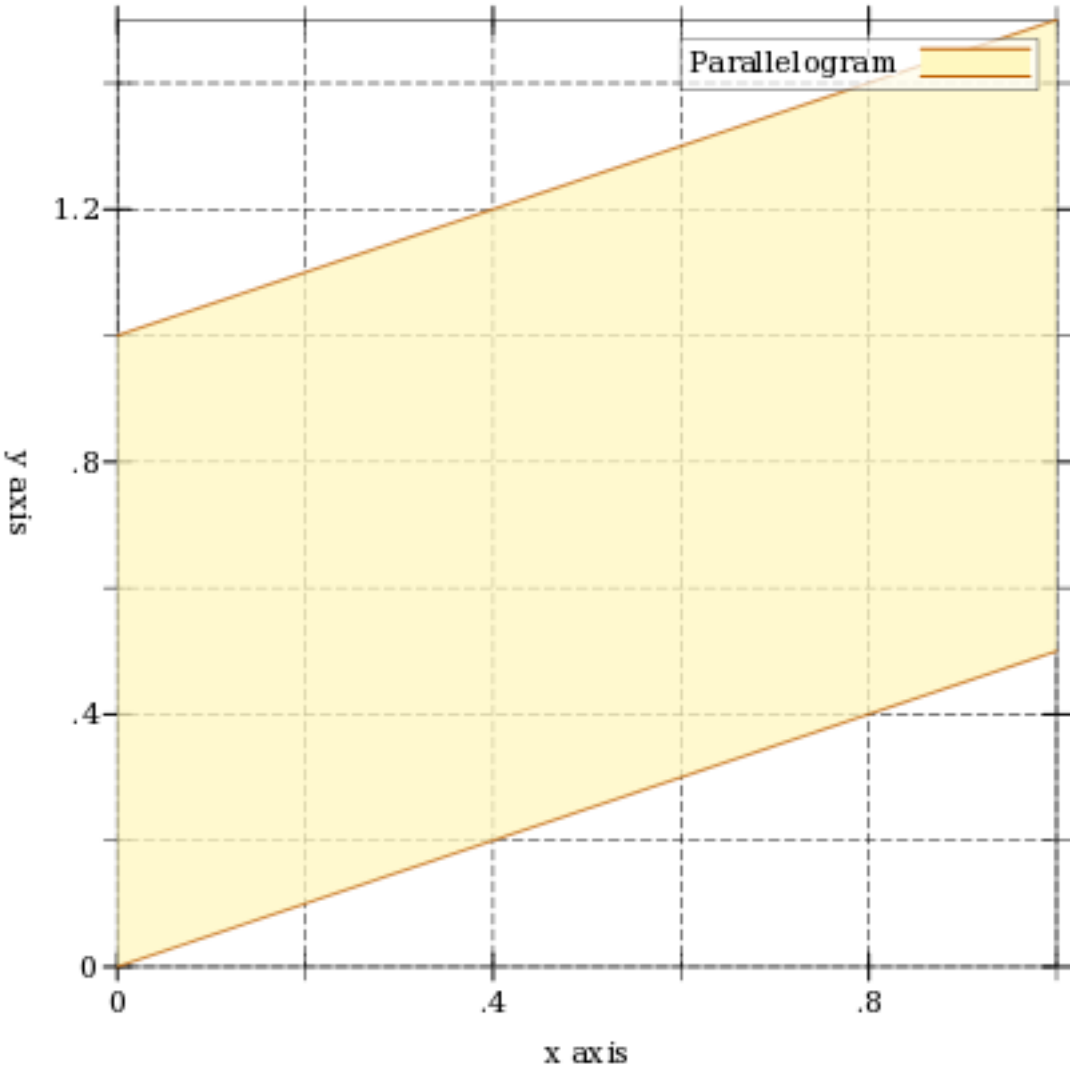
```
> (plot (inverse-interval sin (λ (x) 0) (- pi) pi
                          #:line2-style 'long-dash))
```


2
0
-2
y axis
-.8
-.4
0
.4
.8
x axis

```
(lines-interval  v1s
                 v2s
                [#:x-min x-min
                 #:x-max x-max
                 #:y-min y-min
                 #:y-max y-max
                 #:color color
                 #:style style
                 #:line1-color line1-color
                 #:line1-width line1-width
                 #:line1-style line1-style
                 #:line2-color line2-color
                 #:line2-width line2-width
                 #:line2-style line2-style
                 #:alpha alpha
                 #:label label])           → renderer2d?
  v1s : (listof (vector/c real? real?))
  v2s : (listof (vector/c real? real?))
  x-min : (or/c real? #f) = #f
  x-max : (or/c real? #f) = #f
  y-min : (or/c real? #f) = #f
  y-max : (or/c real? #f) = #f
  color : plot-color/c = (interval-color)
  style : plot-brush-style/c = (interval-style)
  line1-color : plot-color/c = (interval-line1-color)
  line1-width : (>=/c 0) = (interval-line1-width)
  line1-style : plot-pen-style/c = (interval-line1-style)
  line2-color : plot-color/c = (interval-line2-color)
  line2-width : (>=/c 0) = (interval-line2-width)
  line2-style : plot-pen-style/c = (interval-line2-style)
  alpha : (real-in 0 1) = (interval-alpha)
  label : (or/c string? #f) = #f
```

Corresponds with `lines`.

```
> (plot (list
         (tick-grid)
         (lines-interval (list #(0 0) #(1 1/2)) (list #(0 1) #(1 3/2))
                         #:color 4 #:line1-color 4 #:line2-color 4
                         #:label "Parallelogram")))
```

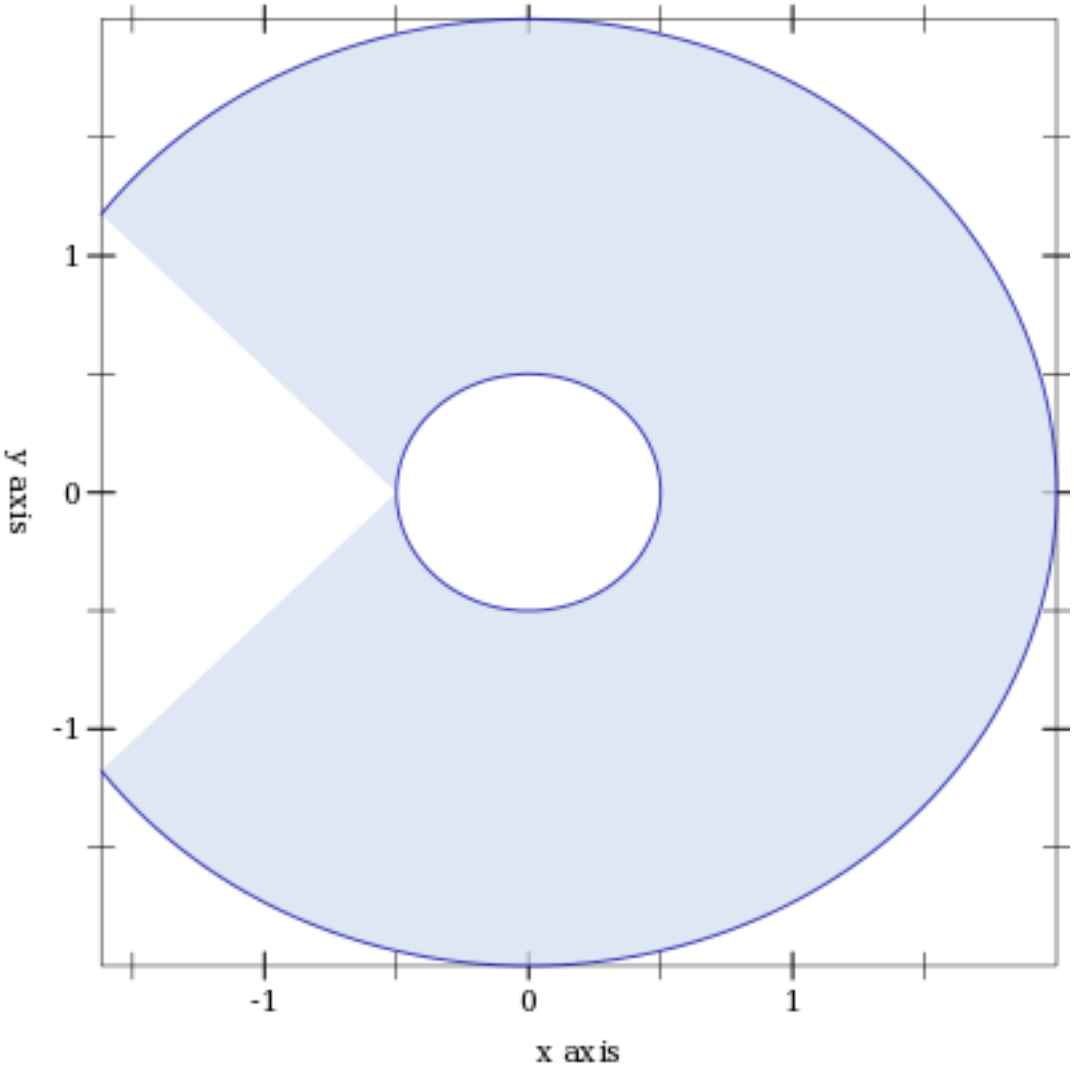

Parallelogram
1.2
.8
.4
0
y axis
0
.4
.8
x axis

```
(parametric-interval  f1
                      f2
                      t-min
                      t-max
                     [#:x-min x-min
                      #:x-max x-max
                      #:y-min y-min
                      #:y-max y-max
                      #:samples samples
                      #:color color
                      #:style style
                      #:line1-color line1-color
                      #:line1-width line1-width
                      #:line1-style line1-style
                      #:line2-color line2-color
                      #:line2-width line2-width
                      #:line2-style line2-style
                      #:alpha alpha
                      #:label label])           → renderer2d?
  f1 : (real? . -> . (vector/c real? real?))
  f2 : (real? . -> . (vector/c real? real?))
  t-min : real?
  t-max : real?
  x-min : (or/c real? #f) = #f
  x-max : (or/c real? #f) = #f
  y-min : (or/c real? #f) = #f
  y-max : (or/c real? #f) = #f
  samples : (and/c exact-integer? (>=/c 2)) = (line-samples)
  color : plot-color/c = (interval-color)
  style : plot-brush-style/c = (interval-style)
  line1-color : plot-color/c = (interval-line1-color)
  line1-width : (>=/c 0) = (interval-line1-width)
  line1-style : plot-pen-style/c = (interval-line1-style)
  line2-color : plot-color/c = (interval-line2-color)
  line2-width : (>=/c 0) = (interval-line2-width)
  line2-style : plot-pen-style/c = (interval-line2-style)
  alpha : (real-in 0 1) = (interval-alpha)
  label : (or/c string? #f) = #f
```

Corresponds with `parametric`.

```
> (let ()
    (define (f1 t) (vector (* 2 (cos (* 4/5 t)))
                           (* 2 (sin (* 4/5 t)))))
    (define (f2 t) (vector (* 1/2 (cos t))
                           (* 1/2 (sin t)))))
```

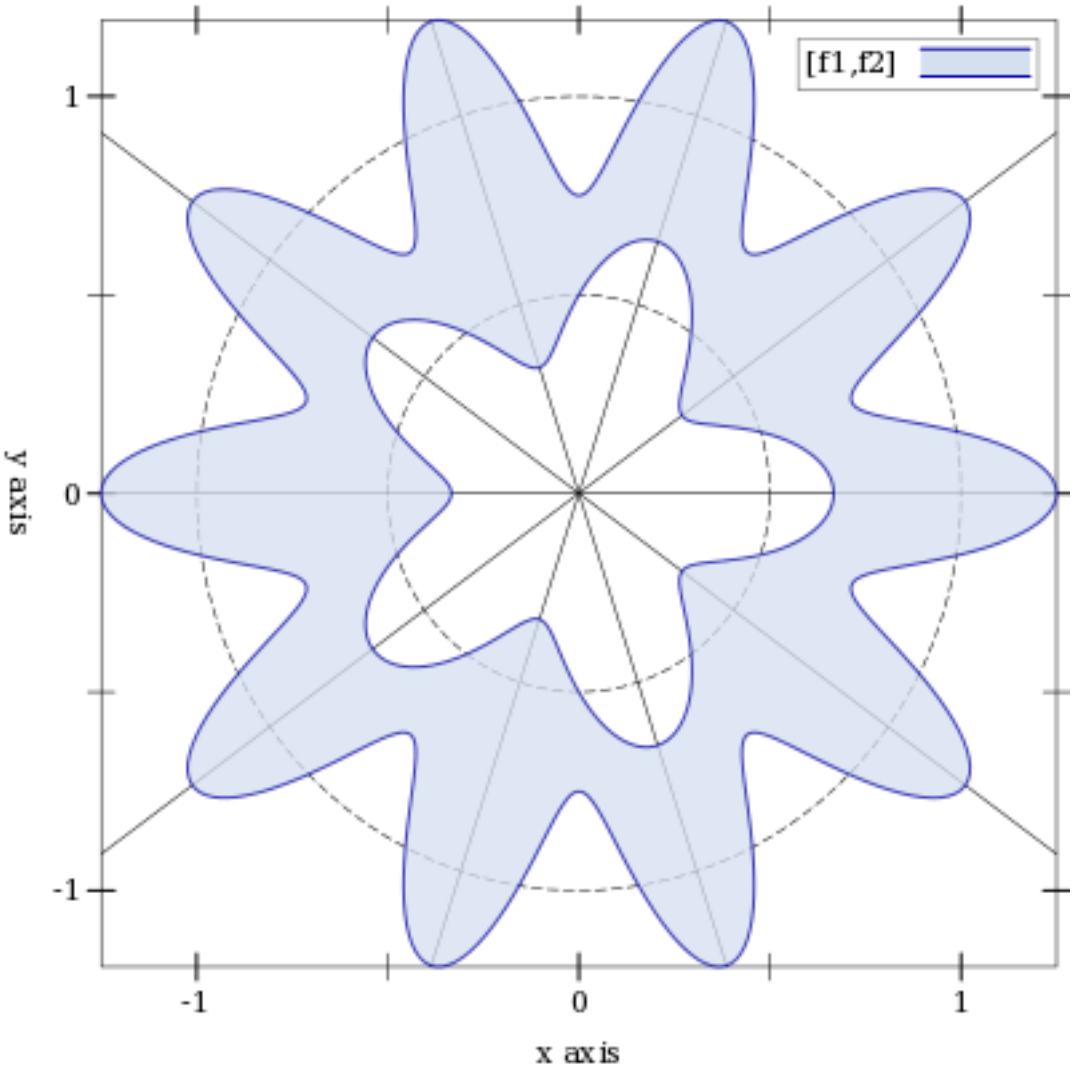
```
(plot (parametric-interval f1 f2 (- pi) pi))
```



```
(polar-interval  f1
                 f2
                [θ-min
                 θ-max
                 #:x-min x-min
                 #:x-max x-max
                 #:y-min y-min
                 #:y-max y-max
                 #:samples samples
                 #:color color
                 #:style style
                 #:line1-color line1-color
                 #:line1-width line1-width
                 #:line1-style line1-style
                 #:line2-color line2-color
                 #:line2-width line2-width
                 #:line2-style line2-style
                 #:alpha alpha
                 #:label label])           → renderer2d?
  f1 : (real? . -> . real?)
  f2 : (real? . -> . real?)
  θ-min : real? = 0
  θ-max : real? = (* 2 pi)
  x-min : (or/c real? #f) = #f
  x-max : (or/c real? #f) = #f
  y-min : (or/c real? #f) = #f
  y-max : (or/c real? #f) = #f
  samples : (and/c exact-integer? (>=/c 2)) = (line-samples)
  color : plot-color/c = (interval-color)
  style : plot-brush-style/c = (interval-style)
  line1-color : plot-color/c = (interval-line1-color)
  line1-width : (>=/c 0) = (interval-line1-width)
  line1-style : plot-pen-style/c = (interval-line1-style)
  line2-color : plot-color/c = (interval-line2-color)
  line2-width : (>=/c 0) = (interval-line2-width)
  line2-style : plot-pen-style/c = (interval-line2-style)
  alpha : (real-in 0 1) = (interval-alpha)
  label : (or/c string? #f) = #f
```

Corresponds with `polar`.

```
> (let ()
    (define (f1 θ) (+ 1/2 (* 1/6 (cos (* 5 θ)))))
    (define (f2 θ) (+ 1 (* 1/4 (cos (* 10 θ)))))
    (plot (list (polar-axes #:number 10)
                (polar-interval f1 f2 #:label "[f1,f2]"))))
```



3.5 2D Contour Renderers

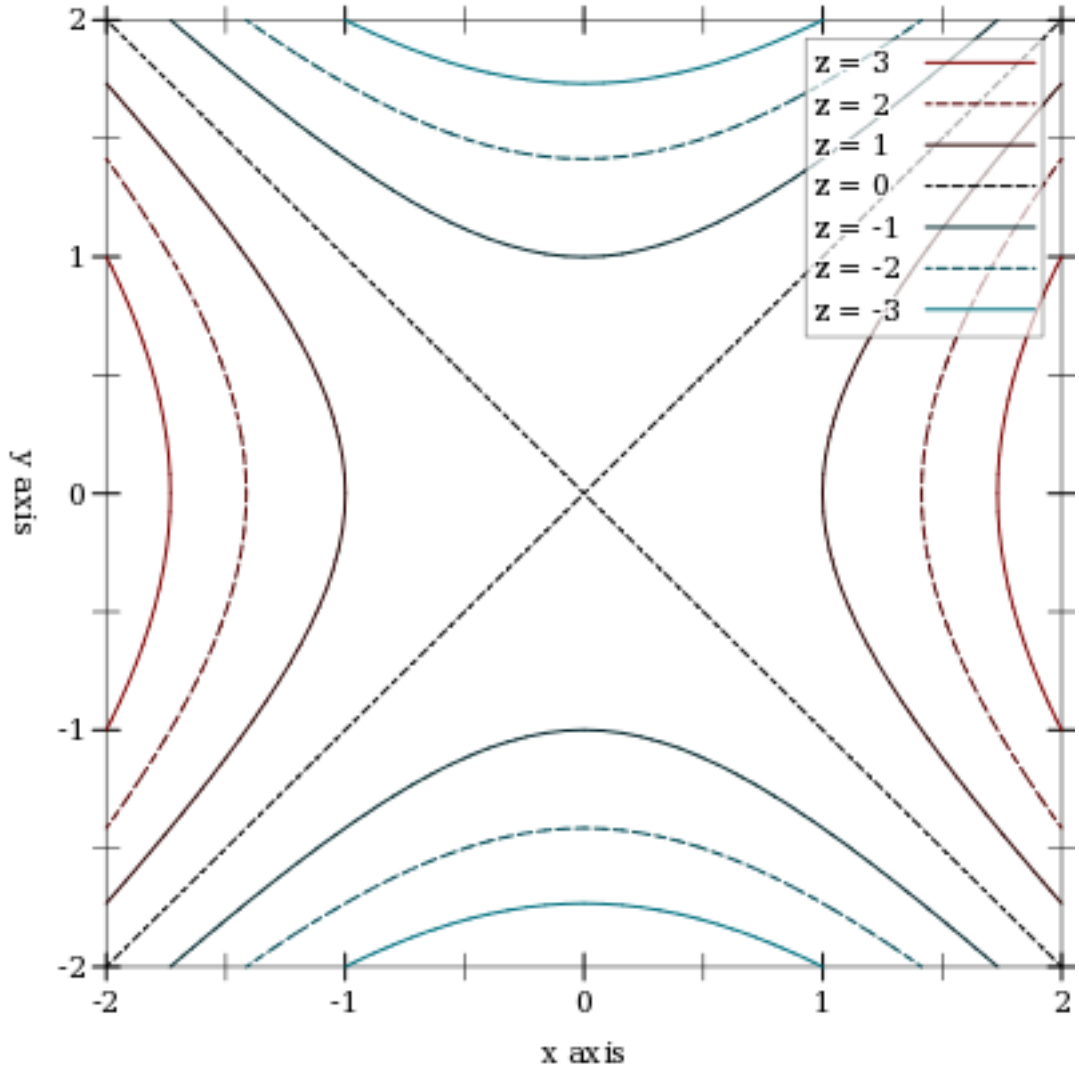
```
(contours  f
           [x-min
            x-max
            y-min
            y-max
            #:levels levels
            #:samples samples
            #:colors colors
            #:widths widths
            #:styles styles
            #:alphas alphas
            #:label label])    → renderer2d?
  f : (real? real? . -> . real?)
  x-min : (or/c real? #f) = #f
  x-max : (or/c real? #f) = #f
  y-min : (or/c real? #f) = #f
  y-max : (or/c real? #f) = #f
  levels : (or/c 'auto exact-positive-integer? (listof real?))
         = (contour-levels)
  samples : (and/c exact-integer? (>=/c 2)) = (contour-samples)
  colors : plot-colors/c = (contour-colors)
  widths : pen-widths/c = (contour-widths)
  styles : plot-pen-styles/c = (contour-styles)
  alphas : alphas/c = (contour-alphas)
  label : (or/c string? #f) = #f
```

Returns a renderer that plots contour lines, or lines of constant height.

When *levels* is 'auto, the number of contour lines and their values are chosen the same way as axis tick positions; i.e. they are chosen to be simple. When *levels* is a number, contours chooses that number of values, evenly spaced, within the output range of *f*. When *levels* is a list, contours plots contours at the values in *levels*.

For example, a saddle:

```
> (plot (contours (λ (x y) (- (sqr x) (sqr y)))
                  -2 2 -2 2 #:label "z"))
```

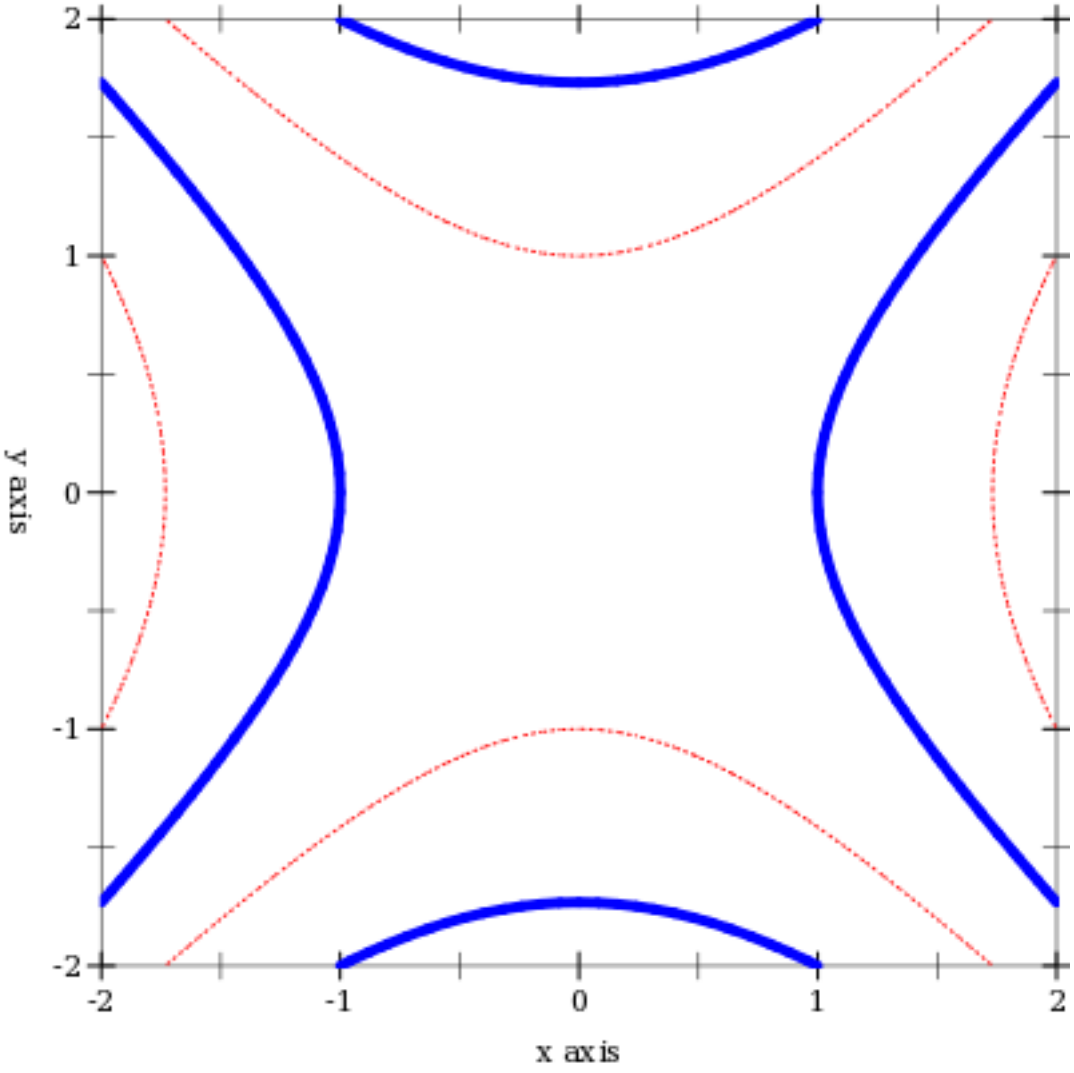


The appearance keyword arguments assign a color, width, style and opacity *to each contour line*. Each can be given as a list or as a function from a list of output values of `f` to a list of appearance values. In both cases, when there are more contour lines than list elements, the colors, widths, styles and alphas in the list repeat.

For example,

```
> (plot (contours (λ (x y) (- (sqr x) (sqr y)))
                  -2 2 -2 2 #:levels 4
                  #:colors '("blue" "red")
```

```
#:widths '(4 1)
#:styles '(solid dot)))
```

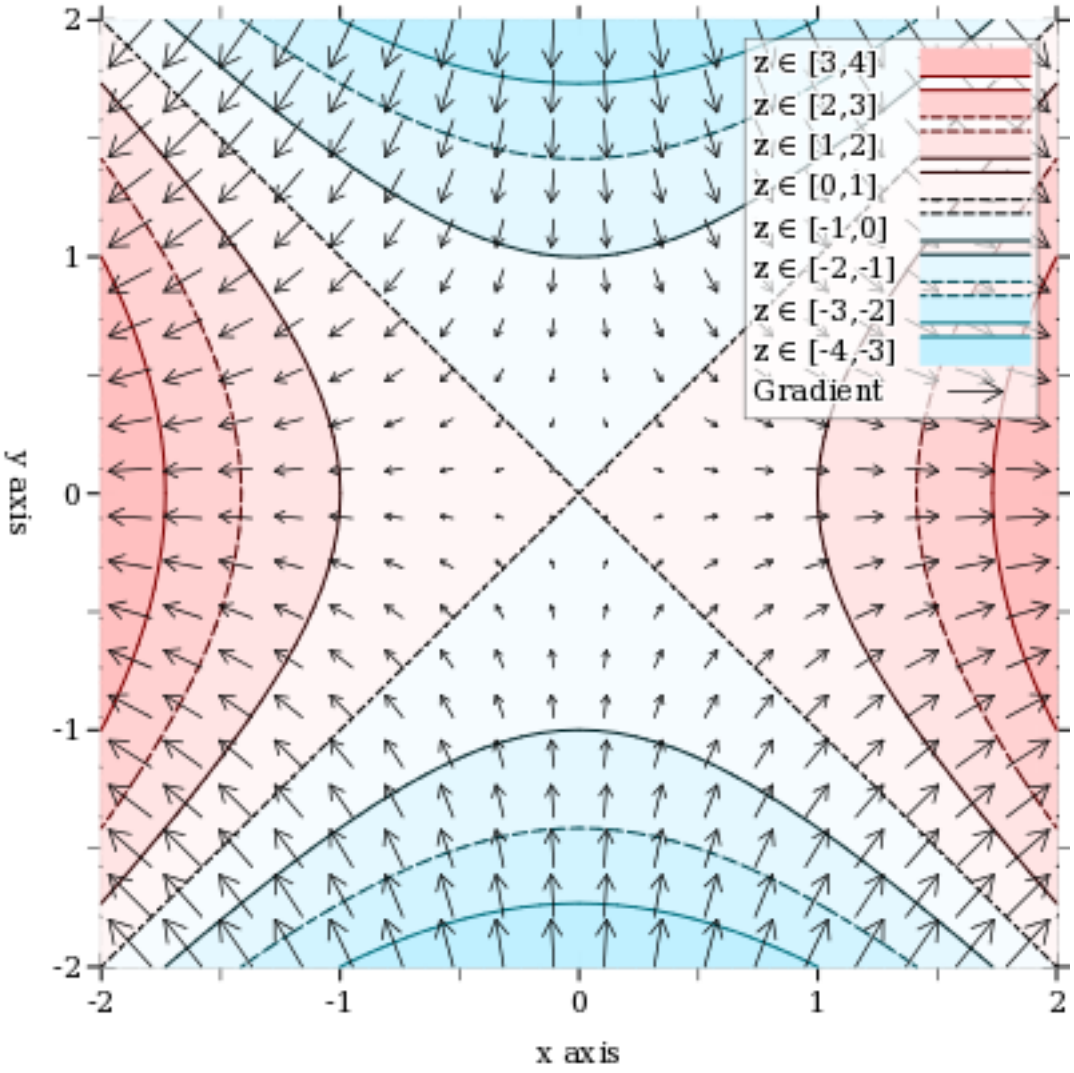


```
(contour-intervals  f
                    [x-min
                     x-max
                     y-min
                     y-max
                     #:levels levels
                     #:samples samples
                     #:colors colors
                     #:styles styles
                     #:contour-colors contour-colors
                     #:contour-widths contour-widths
                     #:contour-styles contour-styles
                     #:alphas alphas
                     #:label label])
 → renderer2d?
  f : (real? real? . -> . real?)
  x-min : (or/c real? #f) = #f
  x-max : (or/c real? #f) = #f
  y-min : (or/c real? #f) = #f
  y-max : (or/c real? #f) = #f
  levels : (or/c 'auto exact-positive-integer? (listof real?))
         = (contour-levels)
  samples : (and/c exact-integer? (>=/c 2)) = (contour-samples)
  colors : plot-colors/c = (contour-interval-colors)
  styles : plot-brush-styles/c = (contour-interval-styles)
  contour-colors : plot-colors/c = (contour-colors)
  contour-widths : pen-widths/c = (contour-widths)
  contour-styles : plot-pen-styles/c = (contour-styles)
  alphas : alphas/c = (contour-interval-alphas)
  label : (or/c string? #f) = #f
```

Returns a renderer that fills the area between contour lines, and additionally draws contour lines.

For example, the canonical saddle, with its gradient field superimposed:

```
> (plot (list (contour-intervals (λ (x y) (- (sqr x) (sqr y)))
                                 -2 2 -2 2 #:label "z")
              (vector-field (λ (x y) (vector (* 2 x) (* -2 y)))
                            #:color "black" #:label "Gradient")))
```



3.6 2D Rectangle Renderers

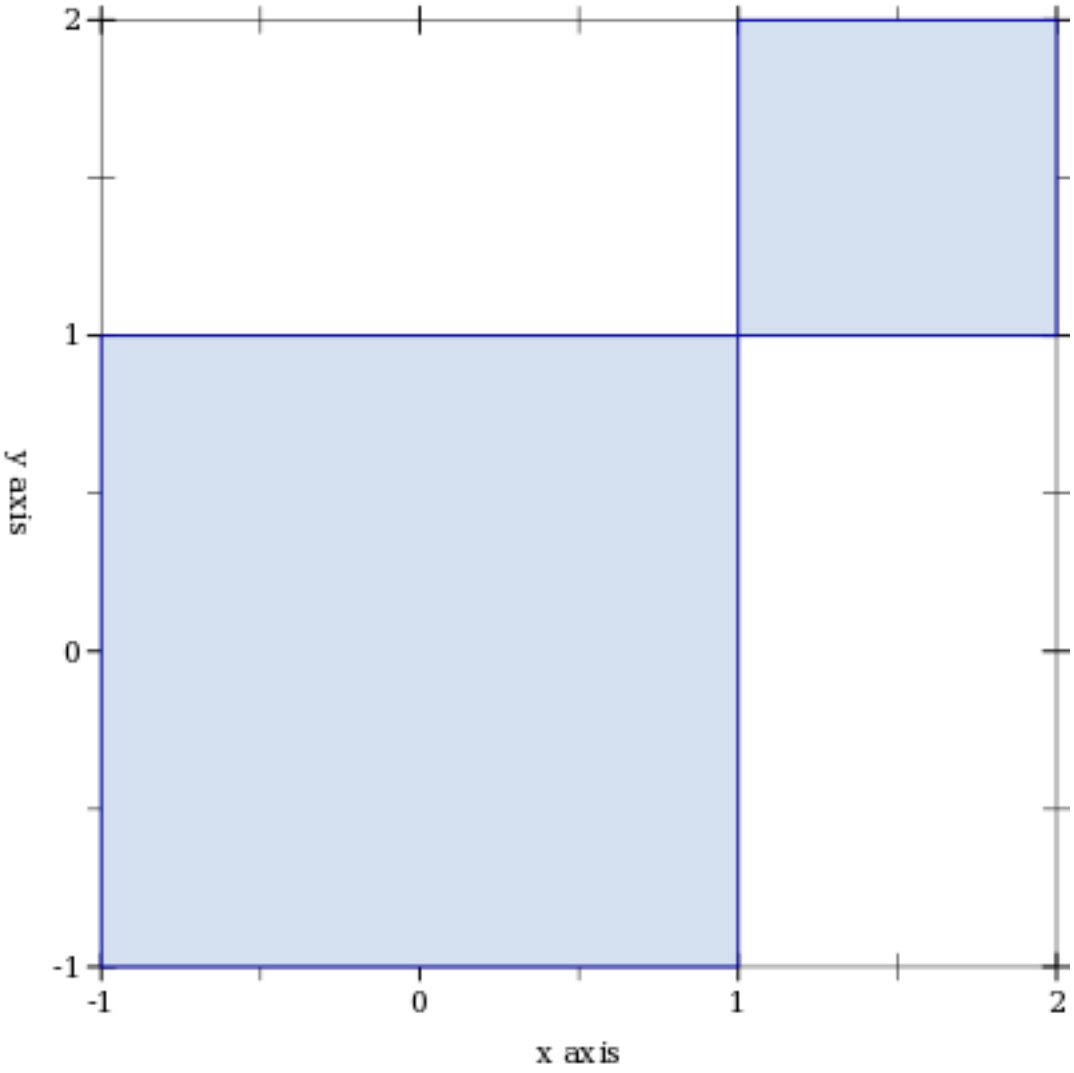
```
(struct ivl (min max)
        #:extra-constructor-name make-ivl)
  min : real?
  max : real?
```

Represents a closed interval. Used to give bounds to rectangles in `rectangles`, `rectangles3d`, and functions derived from them.

```
(rectangles  rects
             [#:x-min x-min
              #:x-max x-max
              #:y-min y-min
              #:y-max y-max
              #:color color
              #:style style
              #:line-color line-color
              #:line-width line-width
              #:line-style line-style
              #:alpha alpha
              #:label label])          → renderer2d?
  rects : (listof (vector/c ivl? ivl?))
  x-min : (or/c real? #f) = #f
  x-max : (or/c real? #f) = #f
  y-min : (or/c real? #f) = #f
  y-max : (or/c real? #f) = #f
  color : plot-color/c = (rectangle-color)
  style : plot-brush-style/c = (rectangle-style)
  line-color : plot-color/c = (rectangle-line-color)
  line-width : (>=/c 0) = (rectangle-line-width)
  line-style : plot-pen-style/c = (rectangle-line-style)
  alpha : (real-in 0 1) = (rectangle-alpha)
  label : (or/c string? #f) = #f
```

Returns a renderer that draws rectangles. The rectanges are given as a list of vectors of intervals—each vector defines the bounds of a rectangle. For example,

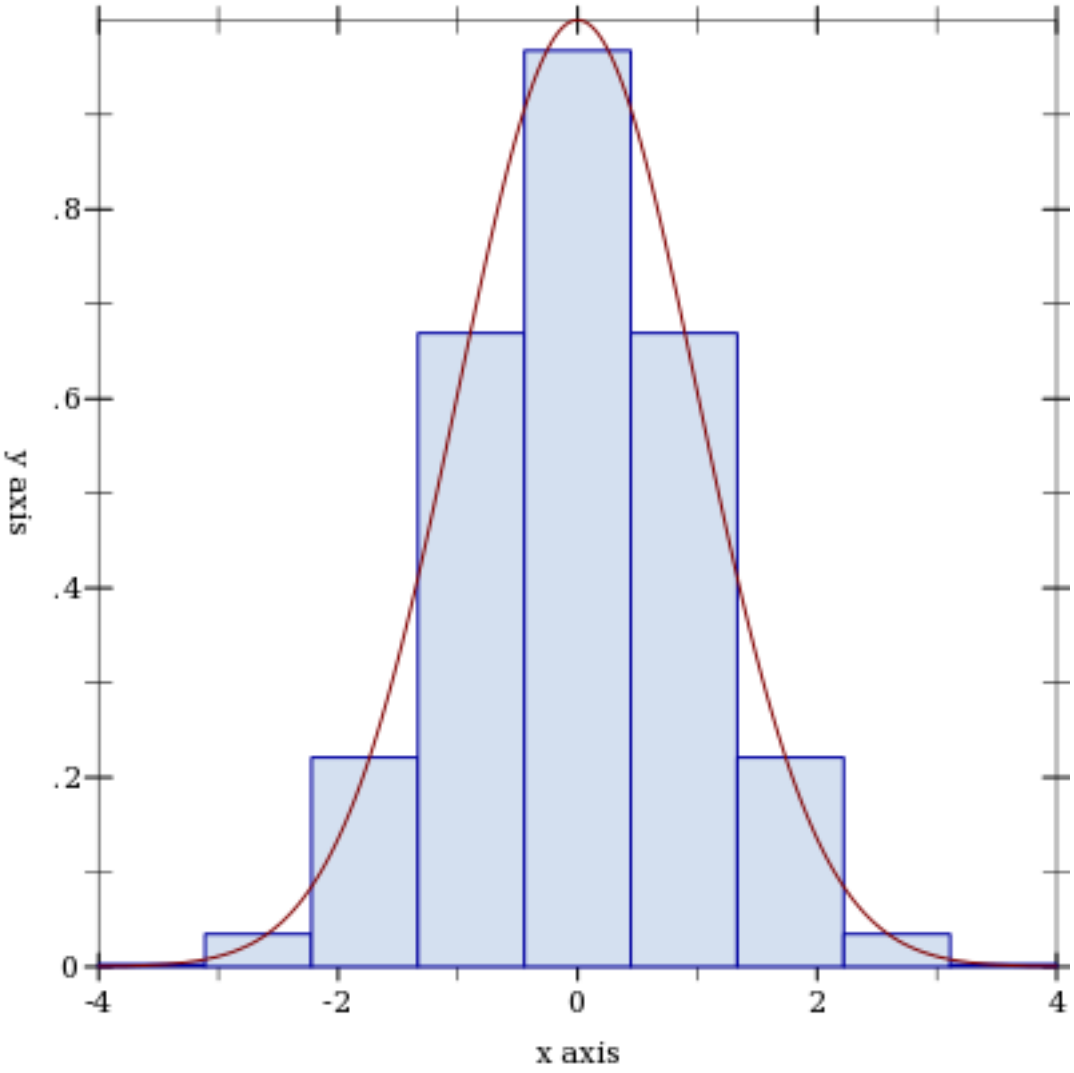
```
> (plot (rectangles (list (vector (ivl -1 1) (ivl -1 1))
                          (vector (ivl 1 2) (ivl 1 2)))))
```


2
1
0
-1
y axis
-1
0
1
2
x axis

```
(area-histogram  f
                 bin-bounds
                [#:x-min x-min
                 #:x-max x-max
                 #:y-min y-min
                 #:y-max y-max
                 #:samples samples
                 #:color color
                 #:style style
                 #:line-color line-color
                 #:line-width line-width
                 #:line-style line-style
                 #:alpha alpha
                 #:label label])          → renderer2d?
  f : (real? . -> . real?)
  bin-bounds : (listof real?)
  x-min : (or/c real? #f) = #f
  x-max : (or/c real? #f) = #f
  y-min : (or/c real? #f) = 0
  y-max : (or/c real? #f) = #f
  samples : (and/c exact-integer? (>=/c 2)) = (line-samples)
  color : plot-color/c = (rectangle-color)
  style : plot-brush-style/c = (rectangle-style)
  line-color : plot-color/c = (rectangle-line-color)
  line-width : (>=/c 0) = (rectangle-line-width)
  line-style : plot-pen-style/c = (rectangle-line-style)
  alpha : (real-in 0 1) = (rectangle-alpha)
  label : (or/c string? #f) = #f
```

Returns a renderer that draws a histogram approximating the area under a curve. The `#:samples` argument determines the accuracy of the calculated areas.

```
> (let ()
    (define (f x) (exp (* -1/2 (sqr x))))
    (plot (list (area-histogram f (linear-seq -4 4 10))
                (function f -4 4))))
```

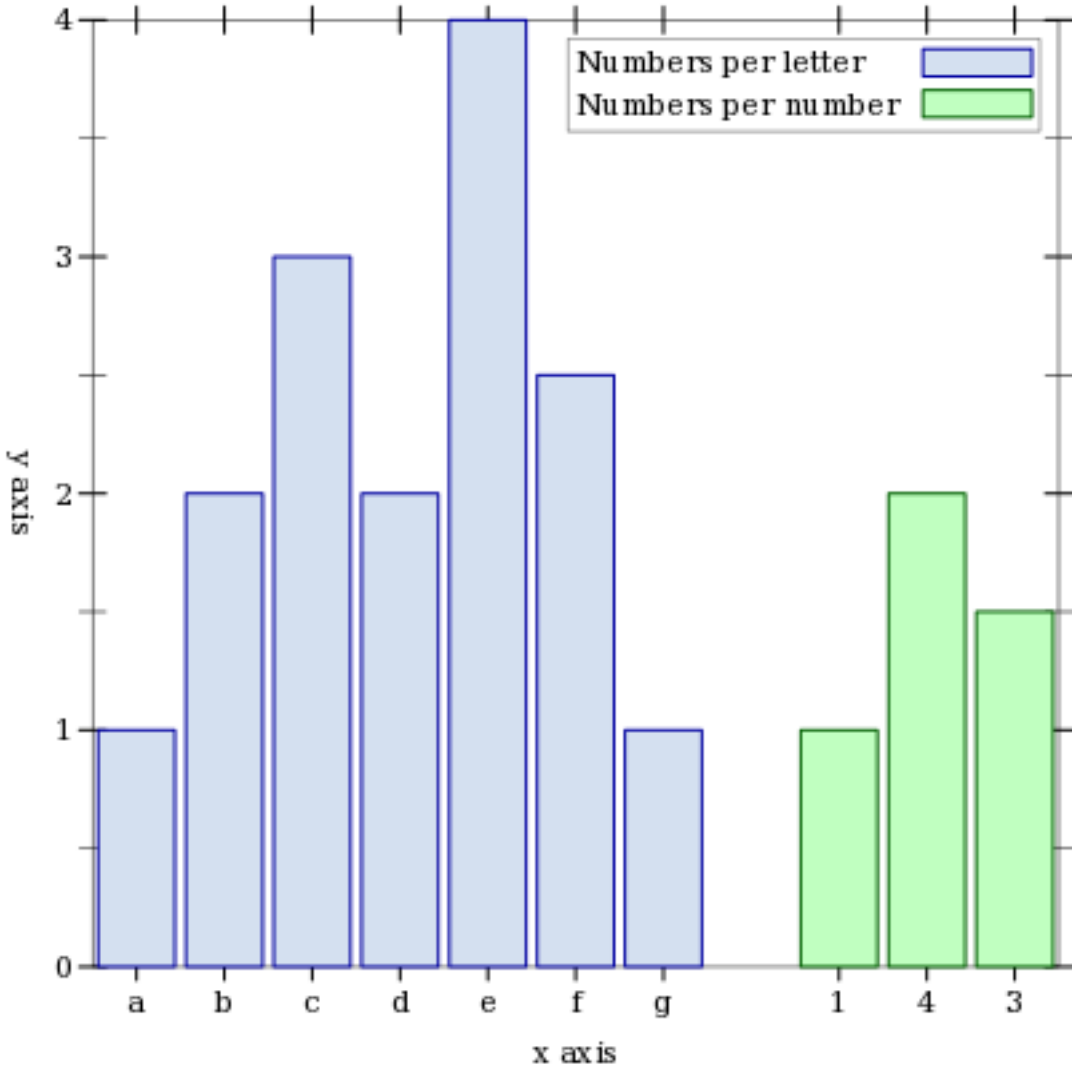

.8
.6
.4
.2
0
y axis
-4
-2
0
2
4
x axis

```
(discrete-histogram  cat-vals
                     [#:x-min x-min
                      #:x-max x-max
                      #:y-min y-min
                      #:y-max y-max
                      #:gap gap
                      #:color color
                      #:style style
                      #:line-color line-color
                      #:line-width line-width
                      #:line-style line-style
                      #:alpha alpha
                      #:label label])          → renderer2d?
  cat-vals : (listof (vector/c any/c real?))
  x-min : (or/c real? #f) = 0
  x-max : (or/c real? #f) = #f
  y-min : (or/c real? #f) = 0
  y-max : (or/c real? #f) = #f
  gap : (real-in 0 1) = (discrete-histogram-gap)
  color : plot-color/c = (rectangle-color)
  style : plot-brush-style/c = (rectangle-style)
  line-color : plot-color/c = (rectangle-line-color)
  line-width : (>=/c 0) = (rectangle-line-width)
  line-style : plot-pen-style/c = (rectangle-line-style)
  alpha : (real-in 0 1) = (rectangle-alpha)
  label : (or/c string? #f) = #f
```

Returns a renderer that draws a discrete histogram.

Each bar takes up exactly one plot unit; e.g. the first bar in a histogram uses the space between `0` and `1`. To plot histograms side-by-side, pass the appropriate `#:x-min` value to the second renderer. For example,

```
> (plot (list (discrete-histogram (list #(a 1) #(b 2) #(c 3) #(d 2)
                                        #(e 4) #(f 2.5) #(g 1))
                                  #:label "Numbers per letter")
              (discrete-histogram (list #(1 1) #(4 2) #(3 1.5))
                                  #:x-min 8
                                  #:color 2 #:line-color 2
                                  #:label "Numbers per number")))
```



3.7 2D Plot Decoration Renderers

```
(x-axis [y #:ticks? ticks?]) → renderer2d?
  y : real? = 0
  ticks? : boolean? = (x-axis-ticks?)
```

Returns a renderer that draws an *x* axis.

```
(y-axis [x #:ticks? ticks?]) → renderer2d?
  x : real? = 0
```

```
  ticks? : boolean? = (y-axis-ticks?)
```

Returns a renderer that draws a *y* axis.

```
(axes [x
       y
       #:x-ticks? x-ticks?
       #:y-ticks? y-ticks?]) → (listof renderer2d?)
  x : real? = 0
  y : real? = 0
  x-ticks? : boolean? = (x-axis-ticks?)
  y-ticks? : boolean? = (y-axis-ticks?)
```

Returns a list containing an `x-axis` renderer and a `y-axis` renderer. See `inverse` for an example.

```
(polar-axes [#:number num #:ticks? ticks?]) → renderer2d?
  num : exact-positive-integer? = (polar-axes-number)
  ticks? : boolean? = (polar-axes-ticks?)
```

Returns a renderer that draws polar axes. See `polar-interval` for an example.

```
(x-tick-lines) → renderer2d?
```

Returns a renderer that draws vertical lines from the lower *x*-axis ticks to the upper.

```
(y-tick-lines) → renderer2d?
```

Returns a renderer that draws horizontal lines from the left *y*-axis ticks to the right.

```
(tick-grid) → (listof renderer2d?)
```

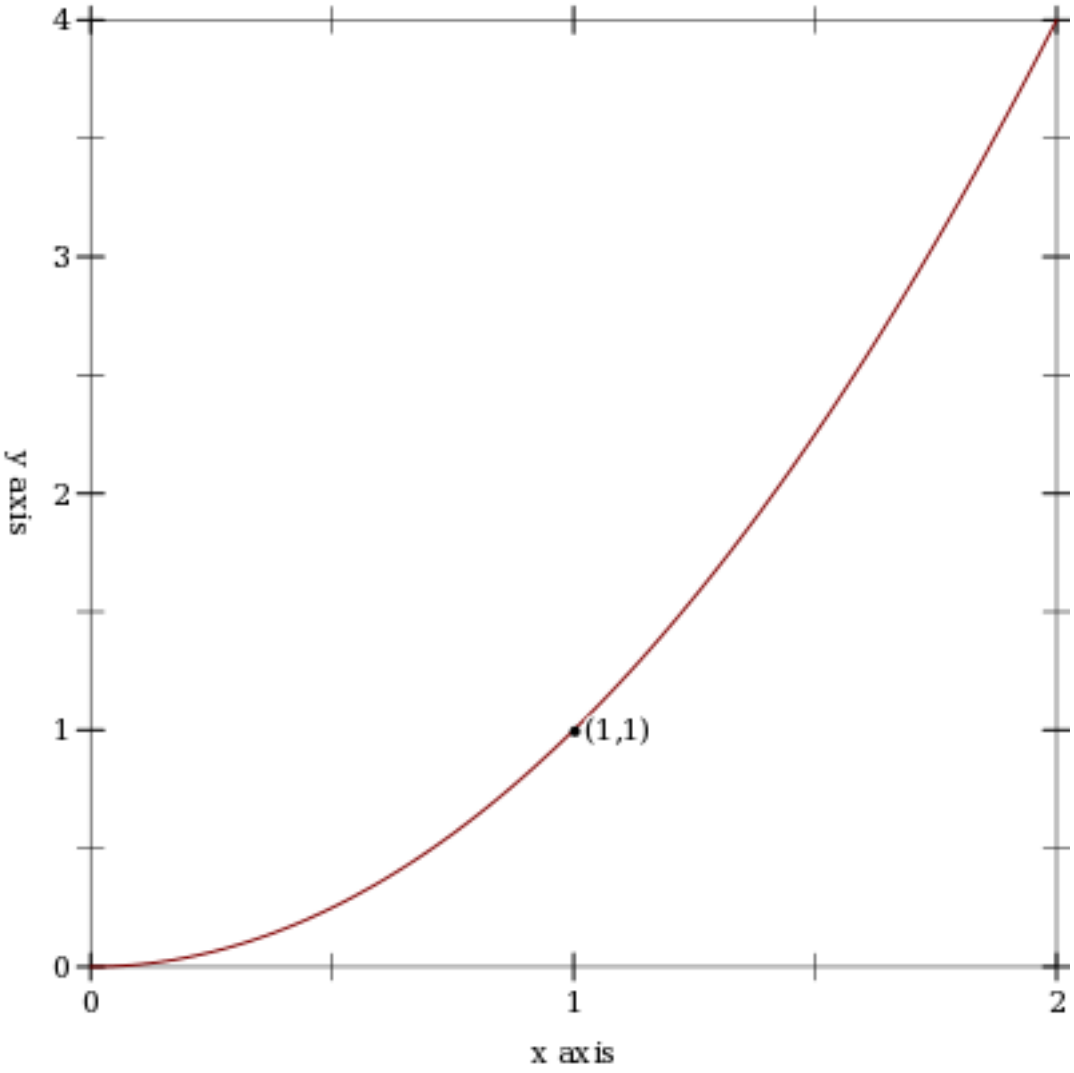
Returns a list containing an `x-tick-lines` renderer and a `y-tick-lines` renderer. See `lines-interval` for an example.

```
(point-label  v
              [label
               #:color color
               #:size size
               #:anchor anchor
               #:angle angle
               #:point-size point-size
               #:alpha alpha])          → renderer2d?
  v : (vector/c real? real?)
```

```
label : (or/c string? #f) = #f
color : plot-color/c = (plot-foreground)
size : (>=/c 0) = (plot-font-size)
anchor : anchor/c = (label-anchor)
angle : real? = (label-angle)
point-size : (>=/c 0) = (label-point-size)
alpha : (real-in 0 1) = (label-alpha)
```

Returns a renderer that draws a labeled point. If *label* is #f, the point is labeled with its position.

```
> (plot (list (function sqr 0 2)
              (point-label (vector 1 1))))
```



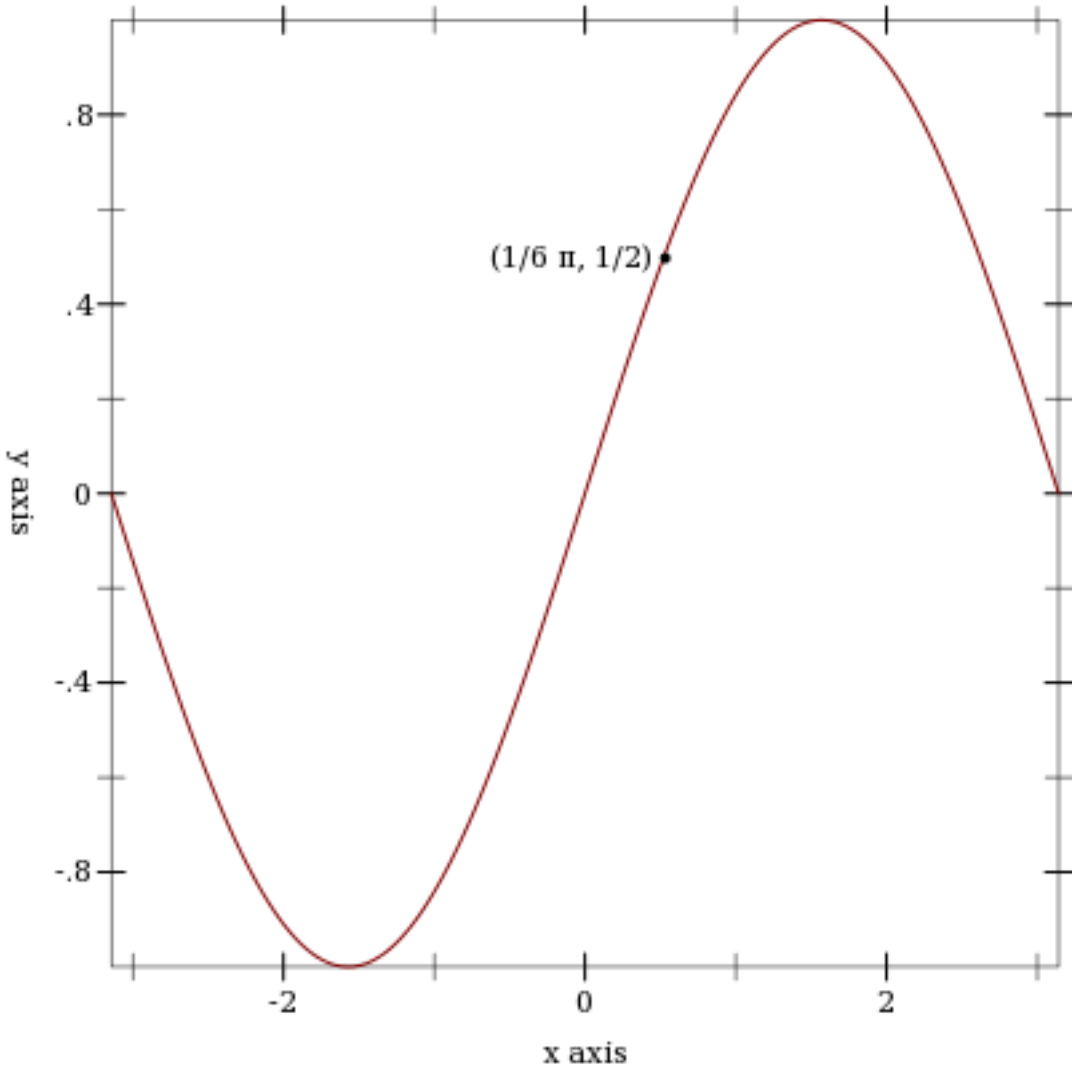
The remaining labeled-point functions are defined in terms of this one.

```
(function-label  f
                 x
                [label
                 #:color color
                 #:size size
                 #:anchor anchor
                 #:angle angle
                 #:point-size point-size
                 #:alpha alpha])          → renderer2d?
```

```
f : (real? . -> . real?)
x : real?
label : (or/c string? #f) = #f
color : plot-color/c = (plot-foreground)
size : (>=/c 0) = (plot-font-size)
anchor : anchor/c = (label-anchor)
angle : real? = (label-angle)
point-size : (>=/c 0) = (label-point-size)
alpha : (real-in 0 1) = (label-alpha)
```

Returns a renderer that draws a labeled point on a function's graph.

```
> (plot (list (function sin (- pi) pi)
              (function-label sin (* 1/6 pi) "(1/6 π, 1/2)"
                              #:anchor 'right)))
```




```
(inverse-label  f
                y
               [label
                #:color color
                #:size size
                #:anchor anchor
                #:angle angle
                #:point-size point-size
                #:alpha alpha])          → renderer2d?
  f : (real? . -> . real?)
  y : real?
```

```
  label : (or/c string? #f) = #f
  color : plot-color/c = (plot-foreground)
  size : (>=/c 0) = (plot-font-size)
  anchor : anchor/c = (label-anchor)
  angle : real? = (label-angle)
  point-size : (>=/c 0) = (label-point-size)
  alpha : (real-in 0 1) = (label-alpha)
```

Returns a renderer that draws a labeled point on a function's inverted graph.

```
(parametric-label  f
                   t
                  [label
                   #:color color
                   #:size size
                   #:anchor anchor
                   #:angle angle
                   #:point-size point-size
                   #:alpha alpha])          → renderer2d?
  f : (real? . -> . (vector/c real? real?))
  t : real?
  label : (or/c string? #f) = #f
  color : plot-color/c = (plot-foreground)
  size : (>=/c 0) = (plot-font-size)
  anchor : anchor/c = (label-anchor)
  angle : real? = (label-angle)
  point-size : (>=/c 0) = (label-point-size)
  alpha : (real-in 0 1) = (label-alpha)
```

Returns a renderer that draws a labeled point on a parametric function's graph.

```
(polar-label  f
              θ
             [label
              #:color color
              #:size size
              #:anchor anchor
              #:angle angle
              #:point-size point-size
              #:alpha alpha])          → renderer2d?
  f : (real? . -> . real?)
  θ : real?
  label : (or/c string? #f) = #f
  color : plot-color/c = (plot-foreground)
  size : (>=/c 0) = (plot-font-size)
```

```
anchor : anchor/c = (label-anchor)
angle : real? = (label-angle)
point-size : (>=/c 0) = (label-point-size)
alpha : (real-in 0 1) = (label-alpha)
```

Returns a renderer that draws a labeled point on a polar function's graph.

4 3D Plot Procedures

Each 3D plot procedure corresponds with a §2 “2D Plot Procedures” procedure. Each behaves the same way as its corresponding 2D procedure, but takes the additional keyword arguments `#:z-min`, `#:z-max`, `#:angle`, `#:altitude` and `#:z-label`.

```
(plot3d  renderer-tree
        [#:x-min x-min
         #:x-max x-max
         #:y-min y-min
         #:y-max y-max
         #:z-min z-min
         #:z-max z-max
         #:width width
         #:height height
         #:angle angle
         #:altitude altitude
         #:title title
         #:x-label x-label
         #:y-label y-label
         #:z-label z-label
         #:legend-anchor legend-anchor
         #:out-file out-file
         #:out-kind out-kind])
 → (or/c (is-a?/c image-snip%) void?)
  renderer-tree : (treeof renderer3d?)
  x-min : (or/c real? #f) = #f
  x-max : (or/c real? #f) = #f
  y-min : (or/c real? #f) = #f
  y-max : (or/c real? #f) = #f
  z-min : (or/c real? #f) = #f
  z-max : (or/c real? #f) = #f
  width : exact-positive-integer? = (plot-width)
  height : exact-positive-integer? = (plot-height)
  angle : real? = (plot3d-angle)
  altitude : real? = (plot3d-altitude)
  title : (or/c string? #f) = (plot-title)
  x-label : (or/c string? #f) = (plot-x-label)
  y-label : (or/c string? #f) = (plot-y-label)
  z-label : (or/c string? #f) = (plot-z-label)
  legend-anchor : anchor/c = (plot-legend-anchor)
  out-file : (or/c path-string? output-port? #f) = #f
  out-kind : (one-of/c 'auto 'png 'jpeg 'xmb 'xpm 'bmp 'ps 'pdf 'svg)
           = 'auto
```

This procedure corresponds with plot. It plots a 3D renderer or list of renderers (or more generally, a tree of renderers), as returned by points3d, parametric3d, surface3d, iso-surface3d, and others.

When the parameter plot-new-window? is #t, plot3d opens a new window to display the plot and returns (void).

When #:out-file is given, plot3d writes the plot to a file using plot3d-file as well as returning a image-snip% or opening a new window.

When given, the *x-min*, *x-max*, *y-min*, *y-max*, *z-min* and *z-max* arguments determine the bounds of the plot, but not the bounds of the renderers.

Deprecated keywords. The #:fgcolor and #:bgcolor keyword arguments are currently supported for backward compatibility, but may not be in the future. Please set the plot-foreground and plot-background parameters instead of using these keyword arguments. The #:lncolor keyword argument is also accepted for backward compatibility but deprecated. It does nothing.

The #:az and #:alt keyword arguments are backward-compatible, deprecated aliases for #:angle and #:altitude, respectively.

```
(plot3d-file  renderer-tree
              output
             [kind]
              #:<plot-keyword> <plot-keyword> ...) → void?
  renderer-tree : (treeof renderer3d?)
  output : (or/c path-string? output-port?)
  kind : (one-of/c 'auto 'png 'jpeg 'xmb 'xpm 'bmp 'ps 'pdf 'svg)
       = 'auto
  <plot-keyword> : <plot-keyword-contract>
(plot3d-pict renderer-tree ...) → pict?
  renderer-tree : (treeof renderer3d?)
(plot3d-bitmap renderer-tree ...) → (is-a?/c bitmap%)
  renderer-tree : (treeof renderer3d?)
(plot3d-snip renderer-tree ...) → (is-a?/c image-snip%)
  renderer-tree : (treeof renderer3d?)
(plot3d-frame renderer-tree ...) → (is-a?/c frame%)
  renderer-tree : (treeof renderer3d?)
```

Plot to different backends. Each of these procedures has the same keyword arguments as plot3d, except for deprecated keywords.

These procedures correspond with plot-file, plot-pict, plot-bitmap, plot-snip and plot-frame.

```
(plot3d/dc  renderer-tree
            dc
            x
            y
            width
            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
            #:angle angle
            #:altitude altitude
            #:title title
            #:x-label x-label
            #:y-label y-label
            #:z-label z-label
            #:legend-anchor legend-anchor]) → void?
  renderer-tree : (treeof renderer3d?)
  dc : (is-a?/c dc<%>)
  x : real?
  y : real?
  width : (>=/c 0)
  height : (>=/c 0)
  x-min : (or/c real? #f) = #f
  x-max : (or/c real? #f) = #f
  y-min : (or/c real? #f) = #f
  y-max : (or/c real? #f) = #f
  z-min : (or/c real? #f) = #f
  z-max : (or/c real? #f) = #f
  angle : real? = (plot3d-angle)
  altitude : real? = (plot3d-altitude)
  title : (or/c string? #f) = (plot-title)
  x-label : (or/c string? #f) = (plot-x-label)
  y-label : (or/c string? #f) = (plot-y-label)
  z-label : (or/c string? #f) = (plot-z-label)
  legend-anchor : anchor/c = (plot-legend-anchor)
```

Plots to an arbitrary device context, in the rectangle with width *width*, height *height*, and upper-left corner *x*,*y*.

Every §4 “3D Plot Procedures” procedure is defined in terms of `plot3d/dc`.

Use this if you need to continually update a plot on a `canvas%`, or to create other `plot3d`-like functions with different backends.

This procedure corresponds with `plot/dc`.

5 3D Renderers

```
(renderer3d? value) → boolean?
  value : any/c
```

Returns `#t` if `value` is a 3D renderer; that is, if `plot3d` can plot `value`. The following functions create such renderers.

5.1 3D Renderer Function Arguments

As with functions that return 2D renderers, functions that return 3D renderers always have these kinds of arguments:

- Required (and possibly optional) arguments representing the graph to plot.
- Optional keyword arguments for overriding calculated bounds, with the default value `#f`.
- Optional keyword arguments that determine the appearance of the plot.
- The optional keyword argument `#:label`, which specifies the name of the renderer in the legend.

See §3.1 “2D Renderer Function Arguments” for a detailed example.

5.2 3D Point Renderers

```
(points3d  vs
           [#:x-min x-min
            #:x-max x-max
            #:y-min y-min
            #:y-max y-max
            #:z-min z-min
            #:z-max z-max
            #:sym sym
            #:color color
            #:size size
            #:line-width line-width
            #:alpha alpha
            #:label label])          → renderer3d?
  vs : (listof (vector/c real? real? real?))
  x-min : (or/c real? #f) = #f
```

```
x-max : (or/c real? #f) = #f
y-min : (or/c real? #f) = #f
y-max : (or/c real? #f) = #f
z-min : (or/c real? #f) = #f
z-max : (or/c real? #f) = #f
sym : point-sym/c = (point-sym)
color : plot-color/c = (point-color)
size : (>=/c 0) = (point-size)
line-width : (>=/c 0) = (point-line-width)
alpha : (real-in 0 1) = (point-alpha)
label : (or/c string? #f) = #f
```

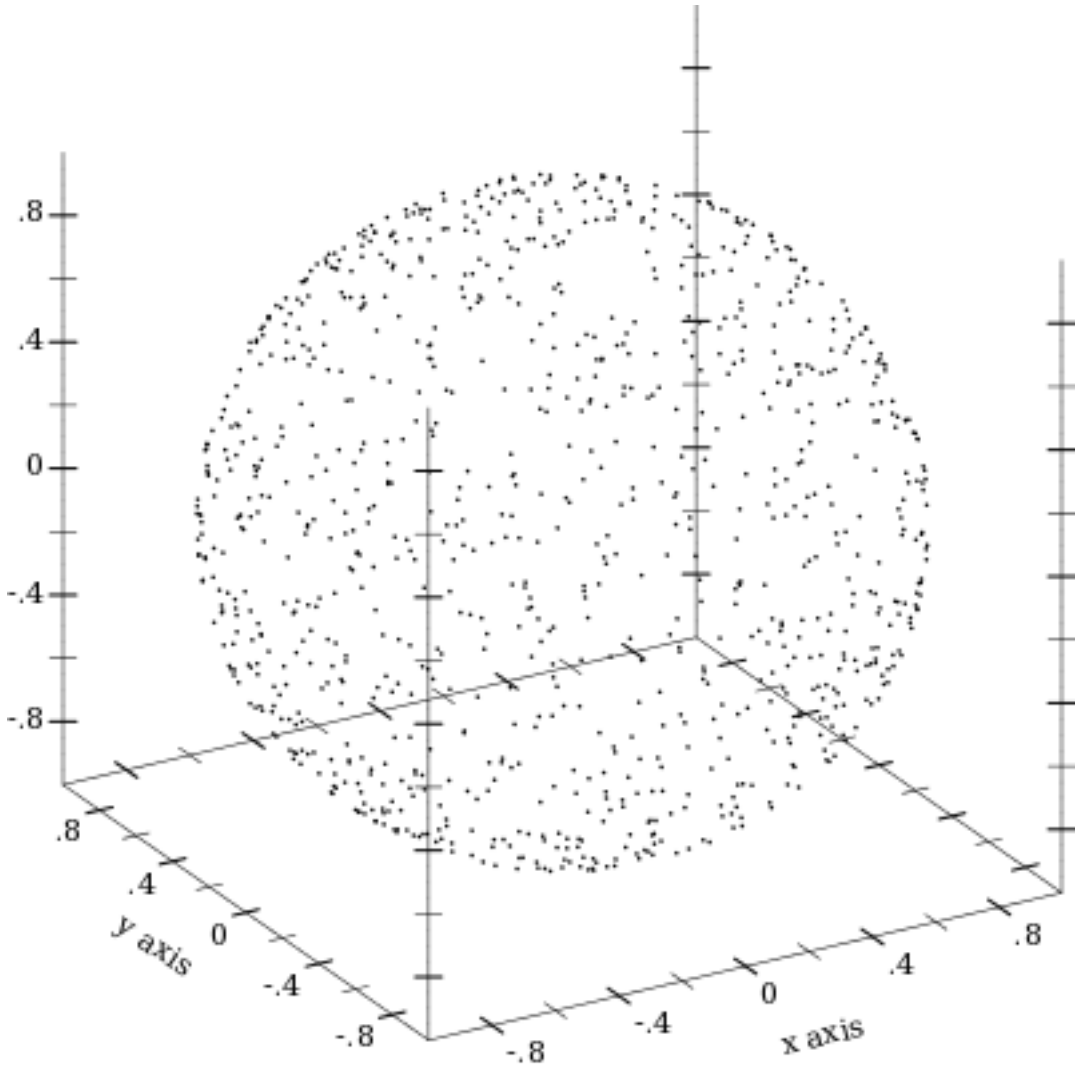
Returns a renderer that draws points in 3D space.

For example, a scatter plot of points sampled uniformly from the surface of a sphere:

```
> (let ()
    (define (runif) (- (* 2 (random)) 1))
    (define (rnormish) (+ (runif) (runif) (runif) (runif)))

    (define xs0 (build-list 1000 (λ _ (rnormish))))
    (define ys0 (build-list 1000 (λ _ (rnormish))))
    (define zs0 (build-list 1000 (λ _ (rnormish))))
    (define mags (map (λ (x y z) (sqrt (+ (sqr x) (sqr y) (sqr z))))
                      xs0 ys0 zs0))
    (define xs (map / xs0 mags))
    (define ys (map / ys0 mags))
    (define zs (map / zs0 mags))

    (plot3d (points3d (map vector xs ys zs) #:sym 'dot)
            #:altitude 25))
```



5.3 3D Line Renderers

```
(lines3d  vs
          [#:x-min x-min
           #:x-max x-max
           #:y-min y-min
           #:y-max y-max
           #:z-min z-min
           #:z-max z-max
           #:color color
           #:width width
           #:style style
           #:alpha alpha
           #:label label]) → renderer3d?
  vs : (listof (vector/c real? real? real?))
  x-min : (or/c real? #f) = #f
  x-max : (or/c real? #f) = #f
  y-min : (or/c real? #f) = #f
  y-max : (or/c real? #f) = #f
  z-min : (or/c real? #f) = #f
  z-max : (or/c real? #f) = #f
  color : plot-color/c = (line-color)
  width : (>=/c 0) = (line-width)
  style : plot-pen-style/c = (line-style)
  alpha : (real-in 0 1) = (line-alpha)
  label : (or/c string? #f) = #f
```

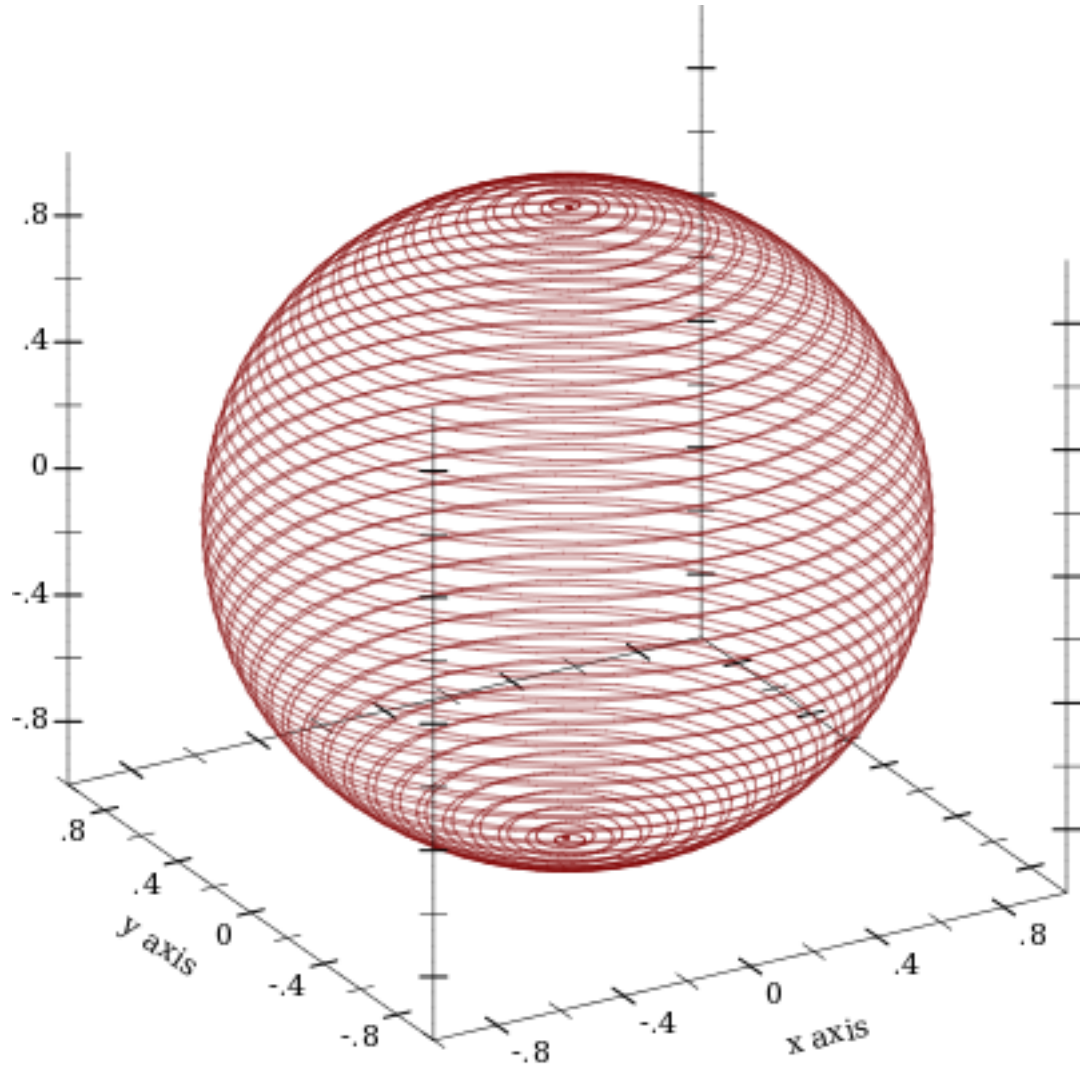
Returns a renderer that draws connected lines, with points in 3D space.

```
(parametric3d  f
               t-min
               t-max
              [#:x-min x-min
               #:x-max x-max
               #:y-min y-min
               #:y-max y-max
               #:z-min z-min
               #:z-max z-max
               #:samples samples
               #:color color
               #:width width
               #:style style
               #:alpha alpha
               #:label label])    → renderer3d?
  f : (real? . -> . (vector/c real? real? real?))
  t-min : real?
  t-max : real?
  x-min : (or/c real? #f) = #f
```

```
x-max : (or/c real? #f) = #f
y-min : (or/c real? #f) = #f
y-max : (or/c real? #f) = #f
z-min : (or/c real? #f) = #f
z-max : (or/c real? #f) = #f
samples : (and/c exact-integer? (>=/c 2)) = (line-samples)
color : plot-color/c = (line-color)
width : (>=/c 0) = (line-width)
style : plot-pen-style/c = (line-style)
alpha : (real-in 0 1) = (line-alpha)
label : (or/c string? #f) = #f
```

Returns a renderer that plots a vector-valued function of time. For example,

```
> (plot3d (parametric3d (λ (t)
                          (vector (* (cos (* 80 t)) (cos t))
                                  (* (sin (* 80 t)) (cos t))
                                  (sin t)))
                        (- pi) pi
                        #:samples 3000 #:alpha 0.5)
          #:altitude 25)
```

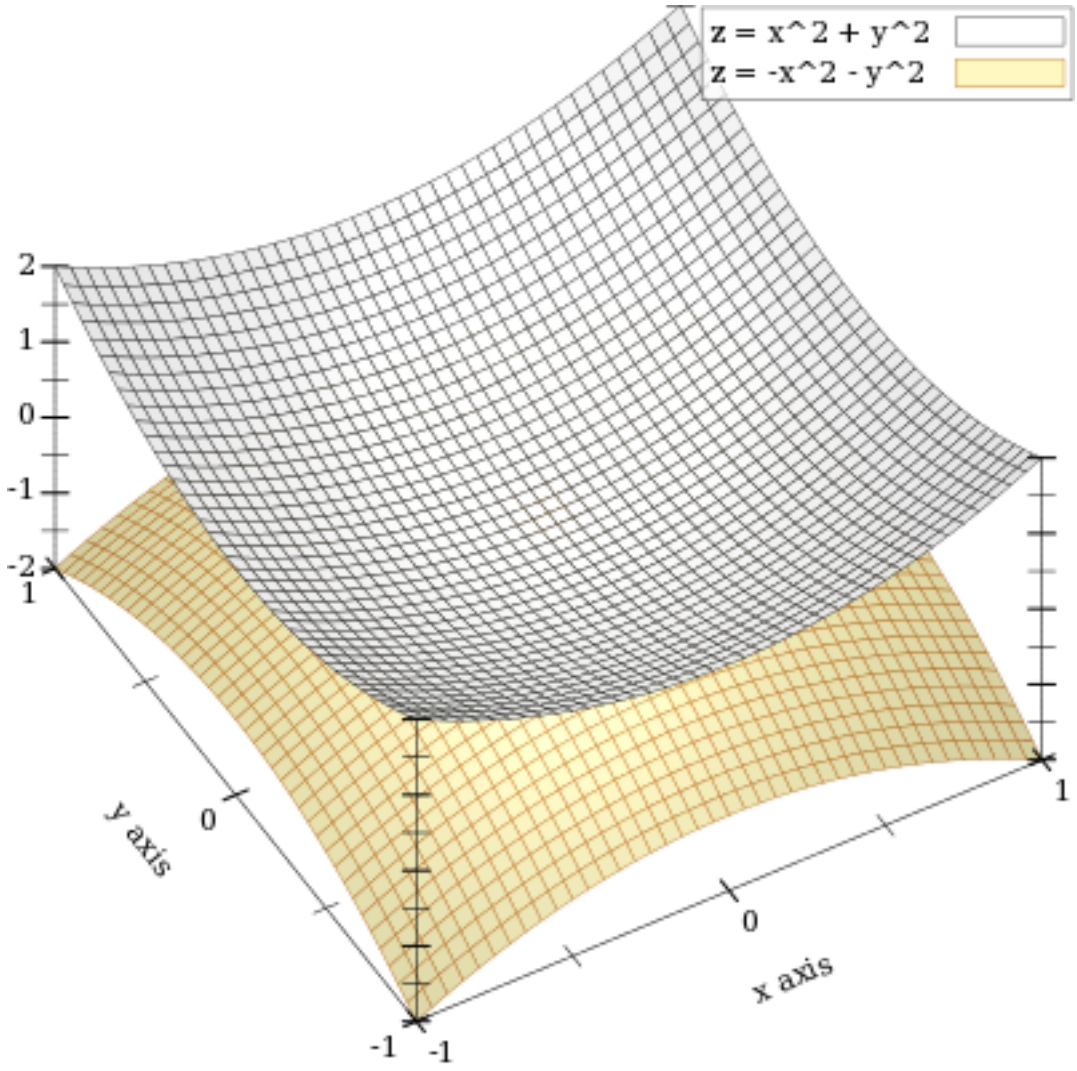


5.4 3D Surface Renderers

```
(surface3d  f
            [x-min
             x-max
             y-min
             y-max
             #:z-min z-min
             #:z-max z-max
             #:samples samples
             #:color color
             #:style style
             #:line-color line-color
             #:line-width line-width
             #:line-style line-style
             #:alpha alpha
             #:label label])          → renderer3d?
  f : (real? real? . -> . real?)
  x-min : (or/c real? #f) = #f
  x-max : (or/c real? #f) = #f
  y-min : (or/c real? #f) = #f
  y-max : (or/c real? #f) = #f
  z-min : (or/c real? #f) = #f
  z-max : (or/c real? #f) = #f
  samples : (and/c exact-integer? (>=/c 2)) = (plot3d-samples)
  color : plot-color/c = (surface-color)
  style : plot-brush-style/c = (surface-style)
  line-color : plot-color/c = (surface-line-color)
  line-width : (>=/c 0) = (surface-line-width)
  line-style : plot-pen-style/c = (surface-line-style)
  alpha : (real-in 0 1) = (surface-alpha)
  label : (or/c string? #f) = #f
```

Returns a renderer that plots a two-input, one-output function. For example,

```
> (plot3d (list (surface3d (λ (x y) (+ (sqr x) (sqr y))) -1 1 -1 1
                           #:label "z = x^2 + y^2")
                (surface3d (λ (x y) (- (+ (sqr x) (sqr y)))) -1 1 -1 1
                           #:color 4 #:line-color 4
                           #:label "z = -x^2 - y^2")))
```


z = x^2 + y^2
z = -x^2 - y^2
2
1
0
-1
-2
1
0
y axis
-1
-1
0
x axis
1

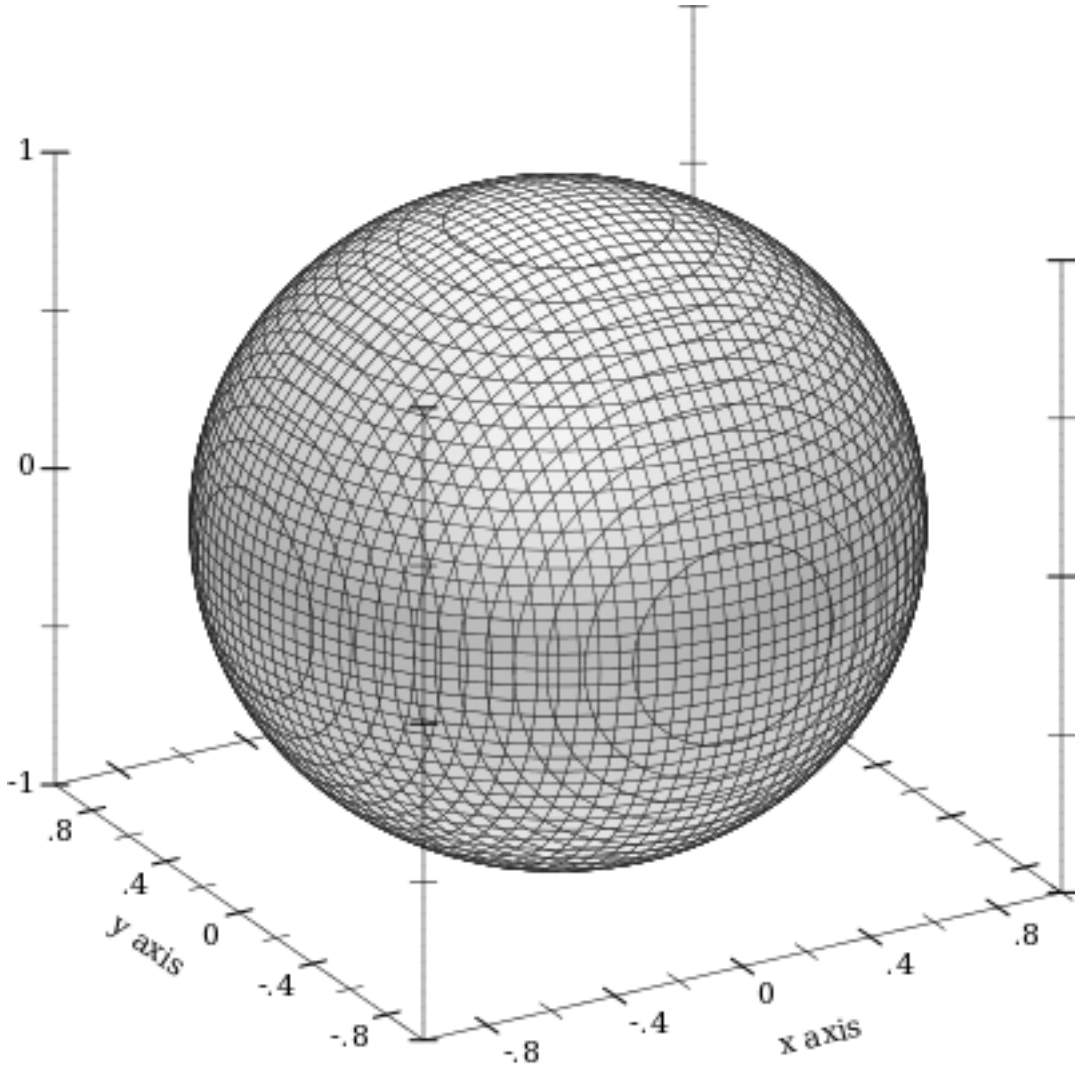
```
(polar3d  f
          [#:x-min x-min
           #:x-max x-max
           #:y-min y-min
           #:y-max y-max
           #:z-min z-min
           #:z-max z-max
           #:samples samples
           #:color color
           #:line-color line-color
           #:line-width line-width
           #:line-style line-style
           #:alpha alpha
           #:label label])          → renderer3d?
  f : (real? real? . -> . real?)
  x-min : (or/c real? #f) = #f
  x-max : (or/c real? #f) = #f
  y-min : (or/c real? #f) = #f
  y-max : (or/c real? #f) = #f
  z-min : (or/c real? #f) = #f
  z-max : (or/c real? #f) = #f
  samples : (and/c exact-integer? (>=/c 2)) = (plot3d-samples)
  color : plot-color/c = (surface-color)
  line-color : plot-color/c = (surface-line-color)
  line-width : (>=/c 0) = (surface-line-width)
  line-style : plot-pen-style/c = (surface-line-style)
  alpha : (real-in 0 1) = (surface-alpha)
  label : (or/c string? #f) = #f
```

Returns a renderer that plots a function from latitude and longitude to radius.

Currently, latitudes range from `0` to `(* 2 pi)`, and longitudes from `(* -1/2 pi)` to `(* 1/2 pi)`.

A sphere is the graph of a polar function of constant radius:

```
> (plot3d (polar3d (λ (θ ρ) 1)) #:altitude 25)
```

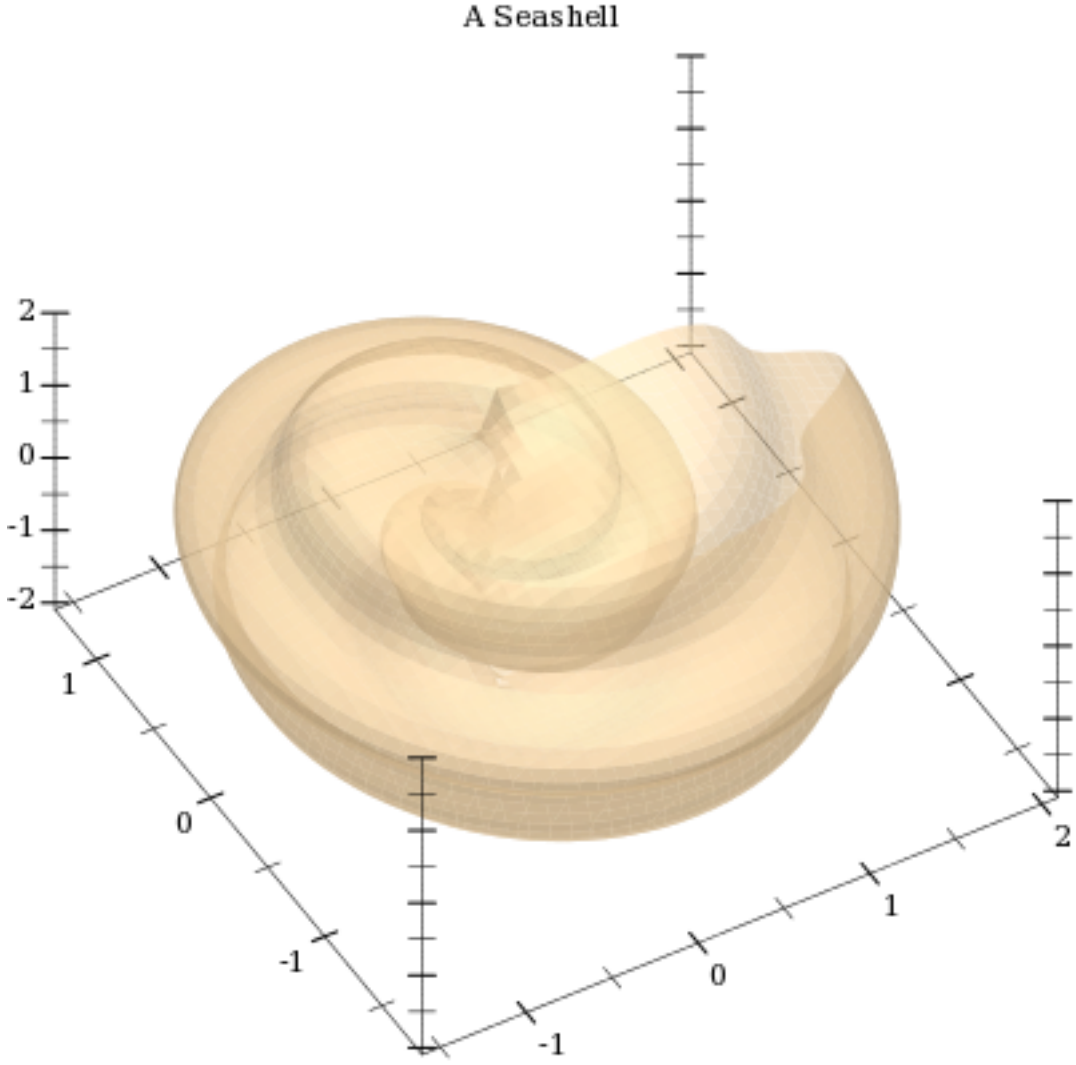


Combining polar function renderers allows faking latitudes or longitudes in larger ranges, to get, for example, a seashell plot:

```
> (let ()
    (define (f1 θ ρ) (+ 1 (/ θ 2 pi) (* 1/8 (sin (* 8 ρ)))))
    (define (f2 θ ρ) (+ 0 (/ θ 2 pi) (* 1/8 (sin (* 8 ρ)))))

    (plot3d (list (polar3d f1 #:color "navajowhite"
                           #:line-style 'transparent #:alpha 2/3)
                  (polar3d f2 #:color "navajowhite"
```

```
                    #:line-style 'transparent #:alpha 2/3))
  #:title "A Seashell" #:x-label #f #:y-label #f))
```



5.5 3D Contour Renderers

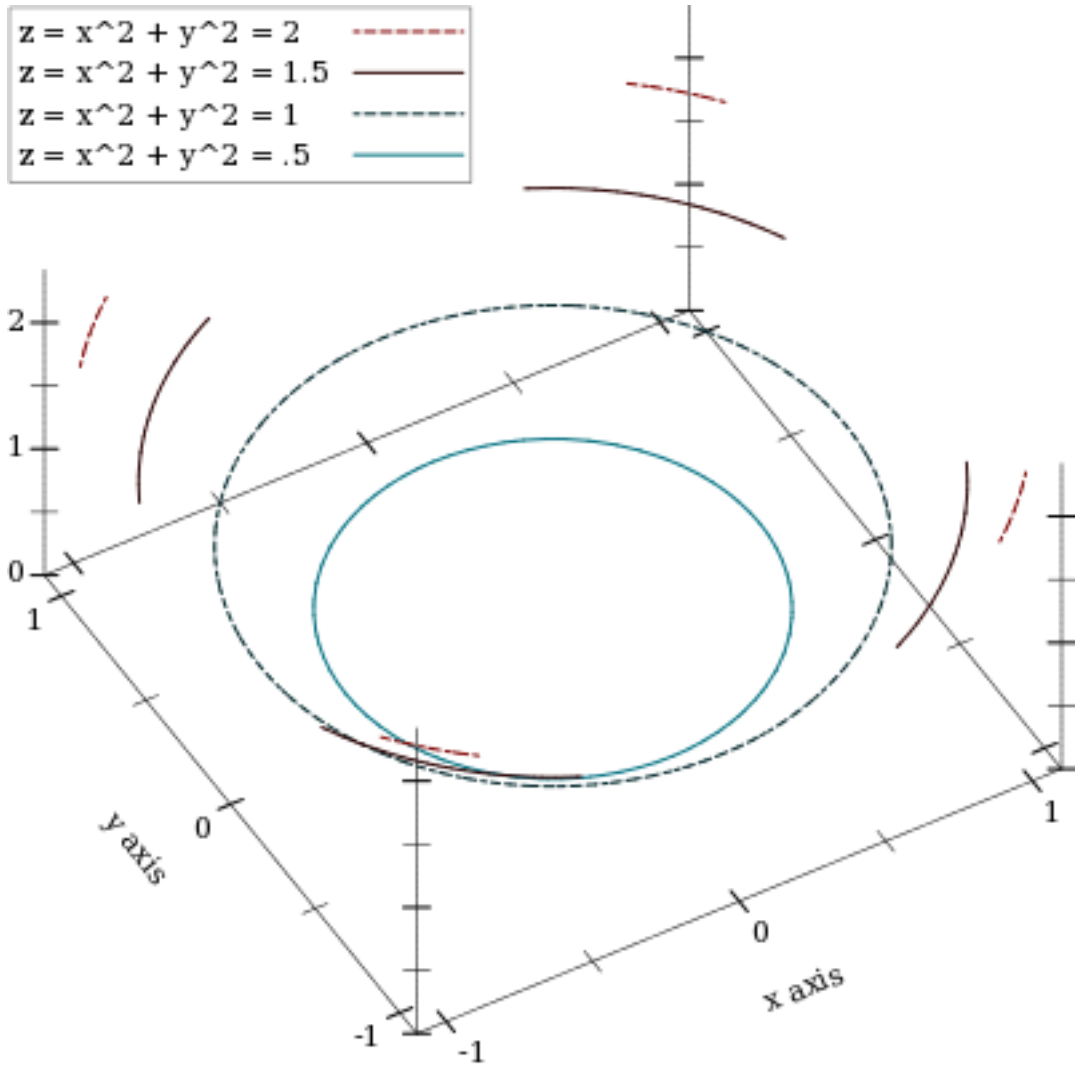
```
(contours3d  f
             [x-min
              x-max
              y-min
              y-max
              #:z-min z-min
              #:z-max z-max
              #:levels levels
              #:samples samples
              #:colors colors
              #:widths widths
              #:styles styles
              #:alphas alphas
              #:label label])    → renderer3d?
  f : (real? real? . -> . real?)
  x-min : (or/c real? #f) = #f
  x-max : (or/c real? #f) = #f
  y-min : (or/c real? #f) = #f
  y-max : (or/c real? #f) = #f
  z-min : (or/c real? #f) = #f
  z-max : (or/c real? #f) = #f
  levels : (or/c 'auto exact-positive-integer? (listof real?))
         = (contour-levels)
  samples : (and/c exact-integer? (>=/c 2)) = (plot3d-samples)
  colors : plot-colors/c = (contour-colors)
  widths : pen-widths/c = (contour-widths)
  styles : plot-pen-styles/c = (contour-styles)
  alphas : alphas/c = (contour-alphas)
  label : (or/c string? #f) = #f
```

Returns a renderer that plots contour lines on the surface of a function.

The appearance keyword arguments are interpreted identically to the appearance keyword arguments to `contours`. In particular, when *levels* is `'auto`, contour values correspond precisely to z axis ticks.

For example,

```
> (plot3d (contours3d (λ (x y) (+ (sqr x) (sqr y))) -1.1 1.1 -1.1 1.1
                      #:label "z = x^2 + y^2")
          #:legend-anchor 'top-left)
```

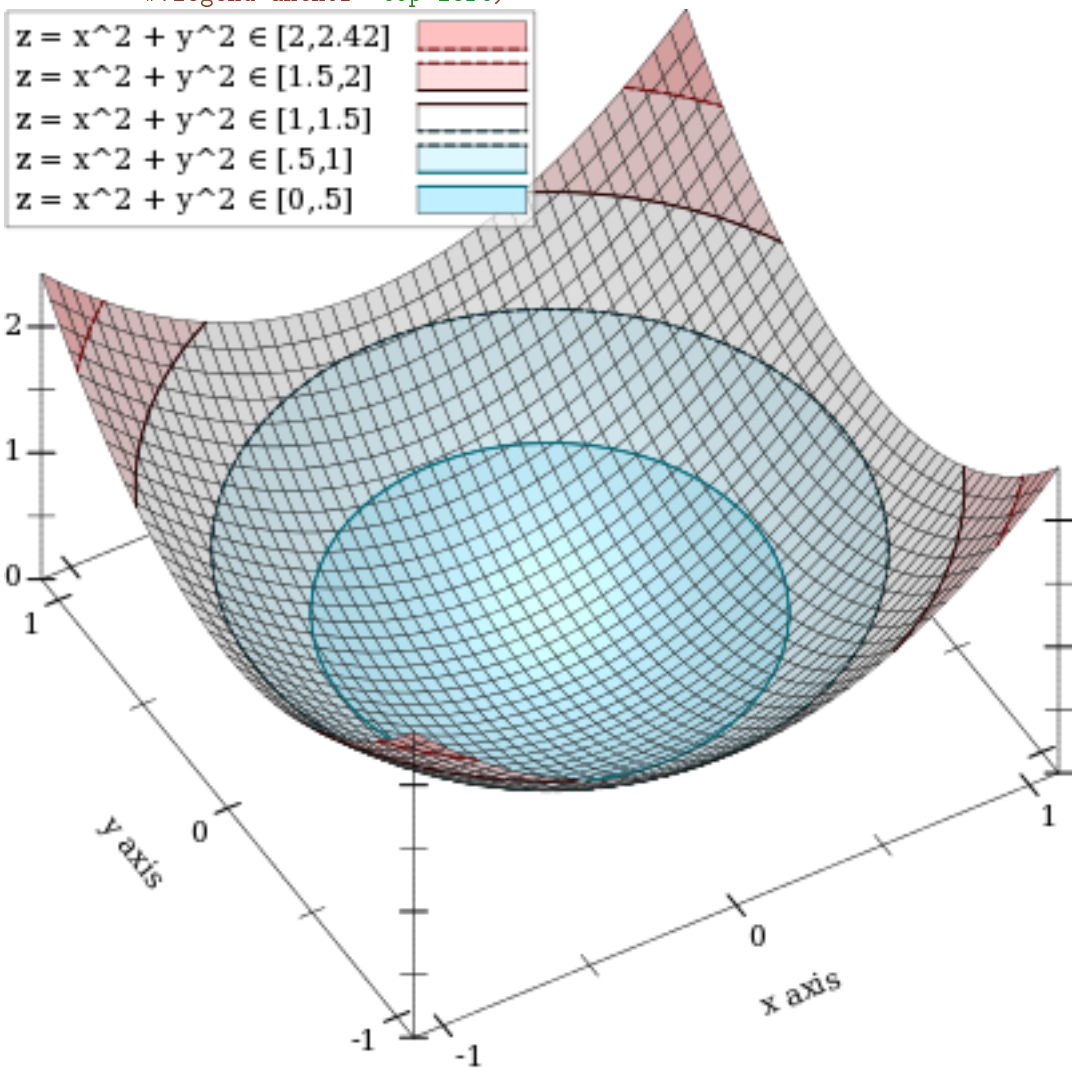

z = x^2 + y^2 = 2
z = x^2 + y^2 = 1.5
z = x^2 + y^2 = 1
z = x^2 + y^2 = .5
2
1
0
1
0
-1
y axis
-1
0
1
x axis

```
(contour-intervals3d  f
                      [x-min
                       x-max
                       y-min
                       y-max
                       #:z-min z-min
                       #:z-max z-max
                       #:levels levels
                       #:samples samples
                       #:colors colors
                       #:line-colors line-colors
                       #:line-widths line-widths
                       #:line-styles line-styles
                       #:contour-colors contour-colors
                       #:contour-widths contour-widths
                       #:contour-styles contour-styles
                       #:alphas alphas
                       #:label label])
 → renderer3d?
  f : (real? real? . -> . real?)
  x-min : (or/c real? #f) = #f
  x-max : (or/c real? #f) = #f
  y-min : (or/c real? #f) = #f
  y-max : (or/c real? #f) = #f
  z-min : (or/c real? #f) = #f
  z-max : (or/c real? #f) = #f
  levels : (or/c 'auto exact-positive-integer? (listof real?))
         = (contour-levels)
  samples : (and/c exact-integer? (>=/c 2)) = (plot3d-samples)
  colors : plot-colors/c = (contour-interval-colors)
  line-colors : plot-colors/c = (contour-interval-line-colors)
  line-widths : pen-widths/c = (contour-interval-line-widths)
  line-styles : plot-pen-styles/c
              = (contour-interval-line-styles)
  contour-colors : plot-colors/c = (contour-colors)
  contour-widths : pen-widths/c = (contour-widths)
  contour-styles : plot-pen-styles/c = (contour-styles)
  alphas : alphas/c = (contour-interval-alphas)
  label : (or/c string? #f) = #f
```

Returns a renderer that plots contour intervals and contour lines on the surface of a function. The appearance keyword arguments are interpreted identically to the appearance keyword arguments to `contour-intervals`.

For example,

```
> (plot3d (contour-intervals3d (λ (x y) (+ (sqr x) (sqr y)))
                               -1.1 1.1 -1.1 1.1
                               #:label "z = x^2 + y^2")
          #:legend-anchor 'top-left)
```



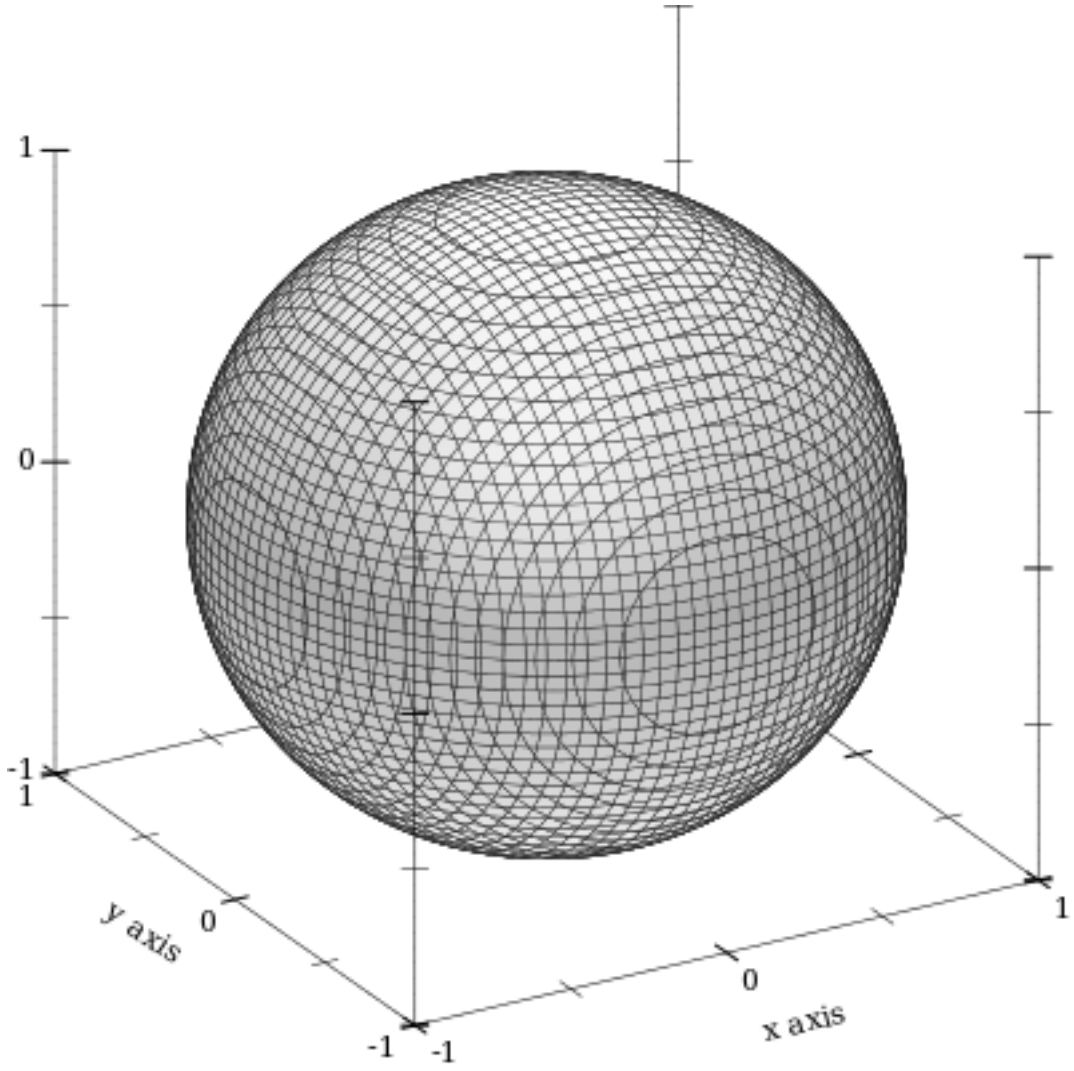
5.6 3D Isosurface Renderers

```
(isosurface3d  f
               d
              [x-min
               x-max
               y-min
               y-max
               z-min
               z-max
               #:samples samples
               #:color color
               #:line-color line-color
               #:line-width line-width
               #:line-style line-style
               #:alpha alpha
               #:label label])          → renderer3d?
  f : (real? real? real? . -> . real?)
  d : real?
  x-min : (or/c real? #f) = #f
  x-max : (or/c real? #f) = #f
  y-min : (or/c real? #f) = #f
  y-max : (or/c real? #f) = #f
  z-min : (or/c real? #f) = #f
  z-max : (or/c real? #f) = #f
  samples : (and/c exact-integer? (>=/c 2)) = (plot3d-samples)
  color : plot-color/c = (surface-color)
  line-color : plot-color/c = (surface-line-color)
  line-width : (>=/c 0) = (surface-line-width)
  line-style : plot-pen-style/c = (surface-line-style)
  alpha : (real-in 0 1) = (surface-alpha)
  label : (or/c string? #f) = #f
```

Returns a renderer that plots the surface of constant output value of the function `f`. The argument `d` is the constant value.

For example, a sphere is all the points in which the Euclidean distance function returns the sphere's radius:

```
> (plot3d (isosurface3d
           (λ (x y z) (sqrt (+ (sqr x) (sqr y) (sqr z)))) 1
           -1 1 -1 1 -1 1)
          #:altitude 25)
```


1
0
-1
1
y axis
0
-1
-1
0
x axis
1

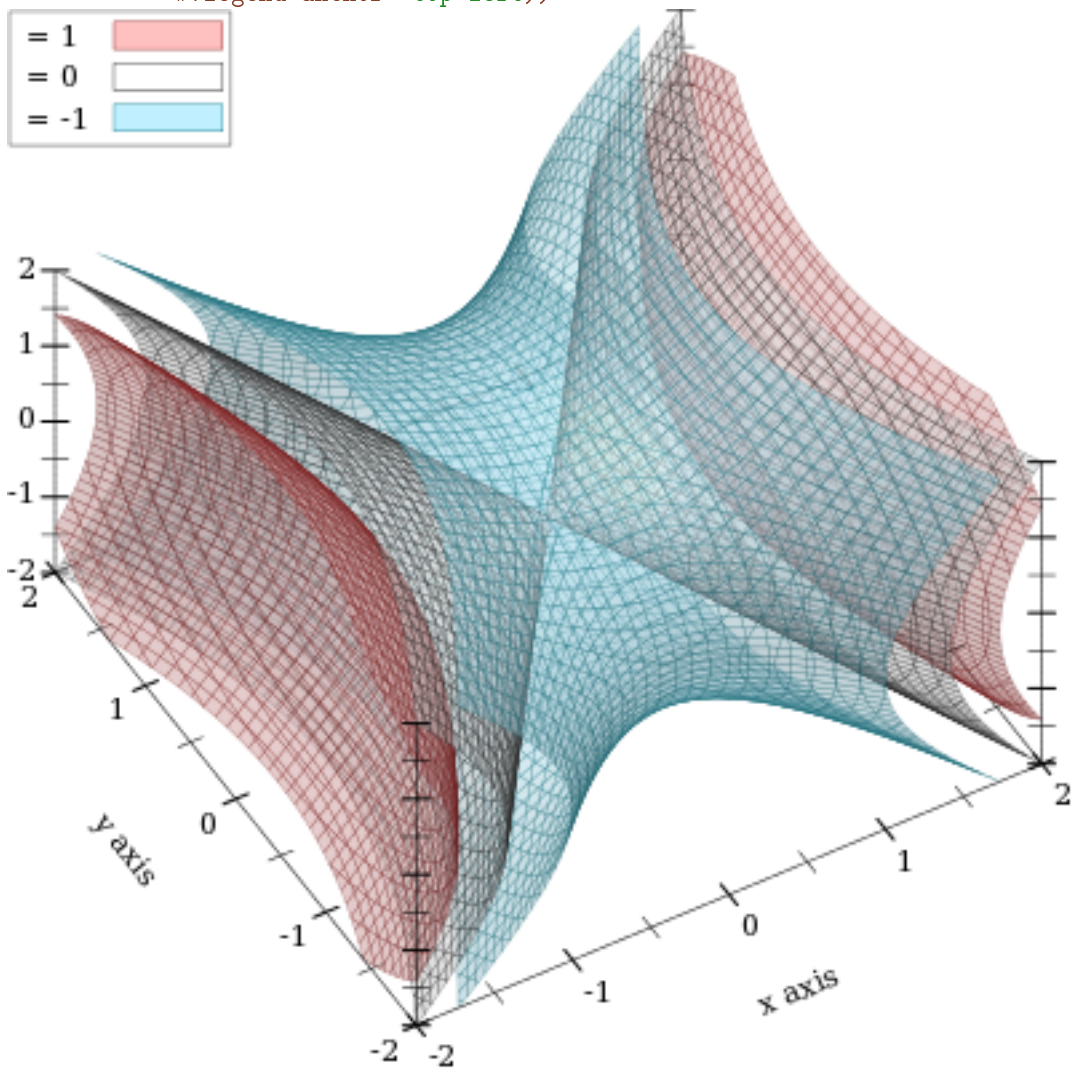
```
(isosurfaces3d  f
                [x-min
                 x-max
                 y-min
                 y-max
                 z-min
                 z-max
                 #:d-min d-min
                 #:d-max d-max
                 #:levels levels
                 #:samples samples
                 #:colors colors
                 #:line-colors line-colors
                 #:line-widths line-widths
                 #:line-styles line-styles
                 #:alphas alphas
                 #:label label])           → renderer3d?
  f : (real? real? real? . -> . real?)
  x-min : (or/c real? #f) = #f
  x-max : (or/c real? #f) = #f
  y-min : (or/c real? #f) = #f
  y-max : (or/c real? #f) = #f
  z-min : (or/c real? #f) = #f
  z-max : (or/c real? #f) = #f
  d-min : (or/c real? #f) = #f
  d-max : (or/c real? #f) = #f
  levels : exact-positive-integer? = (isosurface-levels)
  samples : (and/c exact-integer? (>=/c 2)) = (plot3d-samples)
  colors : plot-colors/c = (isosurface-colors)
  line-colors : plot-colors/c = (isosurface-line-colors)
  line-widths : pen-widths/c = (isosurface-line-widths)
  line-styles : plot-pen-styles/c = (isosurface-line-styles)
  alphas : alphas/c = (isosurface-alphas)
  label : (or/c string? #f) = #f
```

Returns a renderer that plots multiple isosurfaces. The appearance keyword arguments are interpreted similarly to those of `contours`.

Use this to visualize functions from three inputs to one output; for example:

```
> (let ()
    (define (saddle x y z) (- (sqr x) (* 1/2 (+ (sqr y) (sqr z)))))
    (plot3d (isosurfaces3d saddle #:d-min -1 #:d-max 1 #:label "")
            #:x-min -2 #:x-max 2
            #:y-min -2 #:y-max 2
```

```
                #:z-min -2 #:z-max 2
                #:legend-anchor 'top-left))
```



If it helps, think of the output of `f` as a density or charge.

5.7 3D Rectangle Renderers

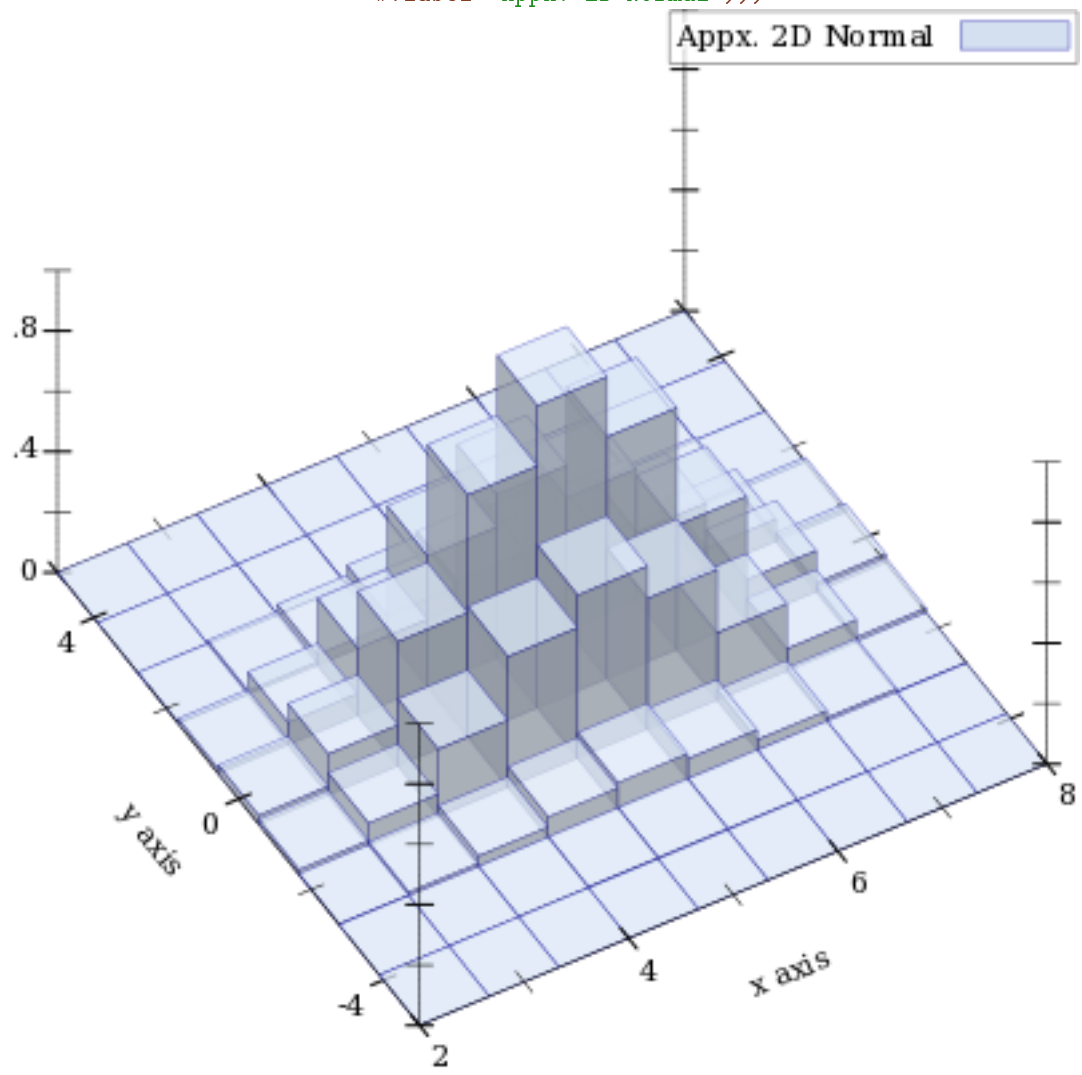
```
(rectangles3d  rects
               [#:x-min x-min
                #:x-max x-max
                #:y-min y-min
                #:y-max y-max
                #:z-min z-min
                #:z-max z-max
                #:color color
                #:style style
                #:line-color line-color
                #:line-width line-width
                #:line-style line-style
                #:alpha alpha
                #:label label])          → renderer3d?
  rects : (listof (vector/c ivl? ivl? ivl?))
  x-min : (or/c real? #f) = #f
  x-max : (or/c real? #f) = #f
  y-min : (or/c real? #f) = #f
  y-max : (or/c real? #f) = #f
  z-min : (or/c real? #f) = #f
  z-max : (or/c real? #f) = #f
  color : plot-color/c = (rectangle-color)
  style : plot-brush-style/c = (rectangle-style)
  line-color : plot-color/c = (rectangle-line-color)
  line-width : (>=/c 0) = (rectangle3d-line-width)
  line-style : plot-pen-style/c = (rectangle-line-style)
  alpha : (real-in 0 1) = (rectangle-alpha)
  label : (or/c string? #f) = #f
```

Returns a renderer that draws rectangles.

This can be used to draw histograms; for example,

```
> (let ()
    (define (norm2 x y) (exp (* -1/2 (+ (sqr (- x 5)) (sqr y)))))
    (define x-ivls (bounds->intervals (linear-seq 2 8 10)))
    (define y-ivls (bounds->intervals (linear-seq -5 5 10)))
    (define x-mids (linear-seq 2 8 9 #:start? #f #:end? #f))
    (define y-mids (linear-seq -5 5 9 #:start? #f #:end? #f))
    (plot3d (rectangles3d (append*
                           (for/list ([y-ivl  (in-list y-ivls)]
                                      [y  (in-list y-mids)])
                             (for/list ([x-ivl  (in-list x-ivls)]
                                        [x  (in-list x-mids)])
                               (define z (norm2 x y))
```

```
      (vector x-ivl y-ivl (ivl 0 z)))))
#:alpha 3/4
#:label "Appx. 2D Normal")))
```

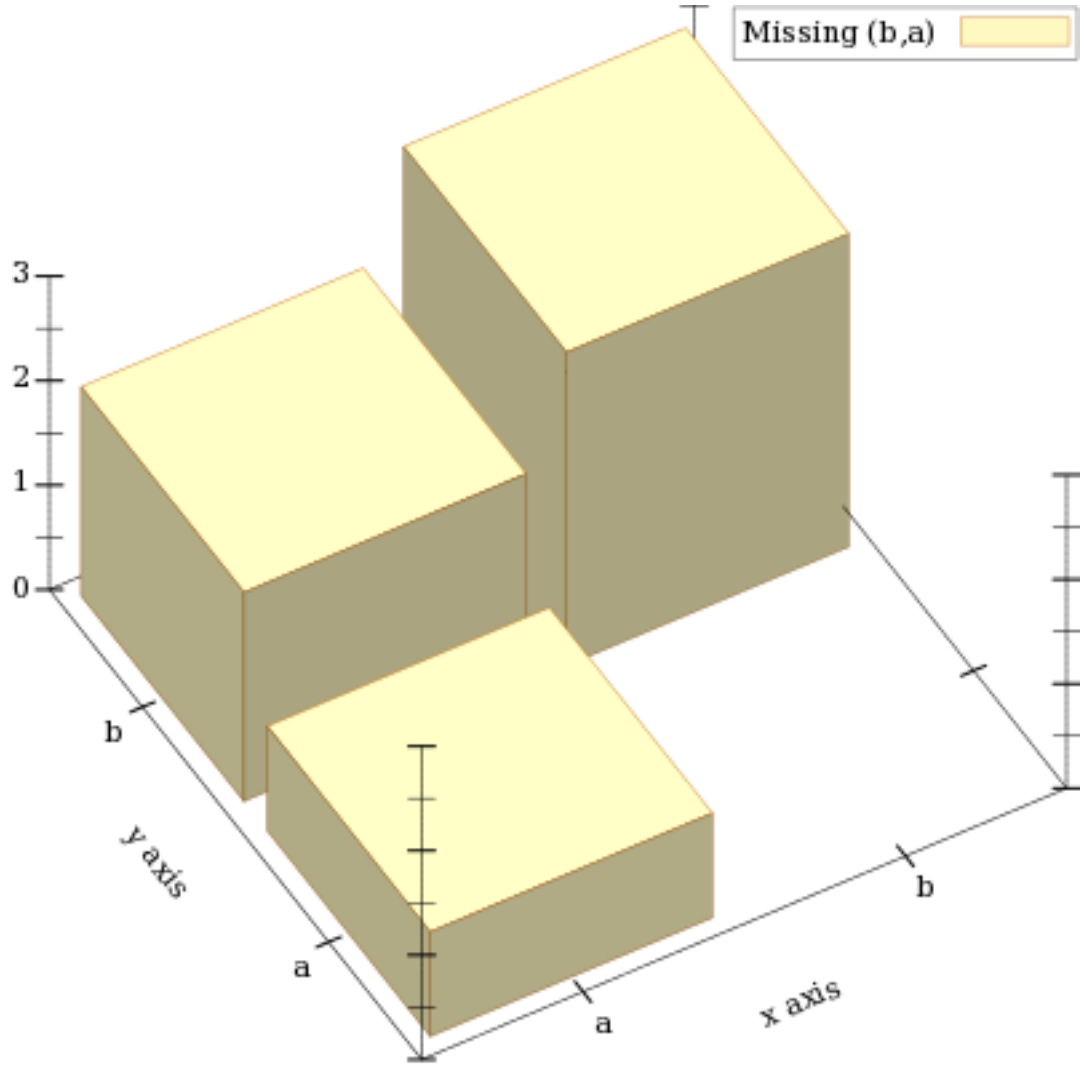


```
(discrete-histogram3d  cat-vals
                       [#:x-min x-min
                        #:x-max x-max
                        #:y-min y-min
                        #:y-max y-max
                        #:z-min z-min
                        #:z-max z-max
                        #:gap gap
                        #:color color
                        #:style style
                        #:line-color line-color
                        #:line-width line-width
                        #:line-style line-style
                        #:alpha alpha
                        #:label label])          → renderer3d?
  cat-vals : (listof (vector/c any/c any/c real?))
  x-min : (or/c real? #f) = 0
  x-max : (or/c real? #f) = #f
  y-min : (or/c real? #f) = 0
  y-max : (or/c real? #f) = #f
  z-min : (or/c real? #f) = 0
  z-max : (or/c real? #f) = #f
  gap : (real-in 0 1) = (discrete-histogram-gap)
  color : plot-color/c = (rectangle-color)
  style : plot-brush-style/c = (rectangle-style)
  line-color : plot-color/c = (rectangle-line-color)
  line-width : (>=/c 0) = (rectangle3d-line-width)
  line-style : plot-pen-style/c = (rectangle-line-style)
  alpha : (real-in 0 1) = (rectangle-alpha)
  label : (or/c string? #f) = #f
```

Returns a renderer that draws discrete histograms on a two-valued domain.

Missing pairs are not drawn; for example,

```
> (plot3d (discrete-histogram3d '(#(a a 1) #(a b 2) #(b b 3))
                                #:label "Missing (b,a)"
                                #:color 4 #:line-color 4))
```


Missing (b,a)
3
2
1
0
b
y axis
a
a
x axis
b

6 Plot Utilities

```
(require plot/utils)
```

```
(degrees->radians d) → real?
  d : real?
```

Converts degrees to radians.

```
(radians->degrees r) → real?
  r : real?
```

Converts radians to degrees.

```
(digits-for-range x-min x-max [extra-digits]) → exact-integer?
  x-min : real?
  x-max : real?
  extra-digits : exact-integer? = 3
```

Given a range, returns the number of decimal places necessary to distinguish numbers in the range. This may return negative numbers for large ranges.

Examples:

```
> (digits-for-range 0.01 0.02)
5
> (digits-for-range 0 100000)
-2
```

```
(real->plot-label x digits) → any
  x : real?
  digits : exact-integer?
```

Converts a real number to a plot label. Used to format axis tick labels, `point-label`s, and numbers in legend entries.

Examples:

```
> (let ([d  (digits-for-range 0.01 0.03)])
    (real->plot-label 0.02555555 d))
".02556"
> (real->plot-label 2352343 -2)
"2352300"
> (real->plot-label 1000000000.0 4)
"1e9"
> (real->plot-label 1000000000.1234 4)
"(1e9)+.1234"
```

```
(->plot-label a [digits]) → string?
  a : any/c
  digits : exact-integer? = 7
```

Converts a Racket value to a label. Used by `discrete-histogram` and `discrete-histogram3d`.

```
(real->string/trunc x e) → string?
  x : real?
  e : exact-integer?
```

Like `real->decimal-string`, but removes trailing zeros and a trailing decimal point.

```
(linear-seq  start
             end
             num
            [#:start? start?
             #:end? end?])    → (listof real?)
  start : real?
  end : real?
  num : exact-nonnegative-integer?
  start? : boolean? = #t
  end? : boolean? = #t
```

Returns a list of evenly spaced real numbers between *start* and *end*. If *start?* is `#t`, the list includes *start*. If *end?* is `#t`, the list includes *end*.

This function is used internally to generate sample points.

Examples:

```
> (linear-seq 0 1 5)
'(0 1/4 1/2 3/4 1)
> (linear-seq 0 1 5 #:start? #f)
'(1/9 1/3 5/9 7/9 1)
> (linear-seq 0 1 5 #:end? #f)
'(0 2/9 4/9 2/3 8/9)
> (linear-seq 0 1 5 #:start? #f #:end? #f)
'(1/10 3/10 1/2 7/10 9/10)
```

```
(linear-seq*  points
              num
             [#:start? start?
              #:end? end?])    → (listof real?)
  points : (listof real?)
```

```
  num : exact-nonnegative-integer?
  start? : boolean? = #t
  end? : boolean? = #t
```

Like linear-seq, but accepts a list of reals instead of a start and end. The #:start? and #:end? keyword arguments work as in linear-seq. This function does not guarantee that each inner value will be in the returned list.

Examples:

```
> (linear-seq* '(0 1 2) 5)
'(0 1/2 1 3/2 2)
> (linear-seq* '(0 1 2) 6)
'(0 2/5 4/5 6/5 8/5 2)
> (linear-seq* '(0 1 0) 5)
'(0 1/2 1 1/2 0)
```

```
(bounds->intervals xs) → (listof ivl?)
  xs : (listof real?)
```

Given a list of points, returns intervals between each pair.

Use this to construct inputs for rectangles and rectangles3d.

Example:

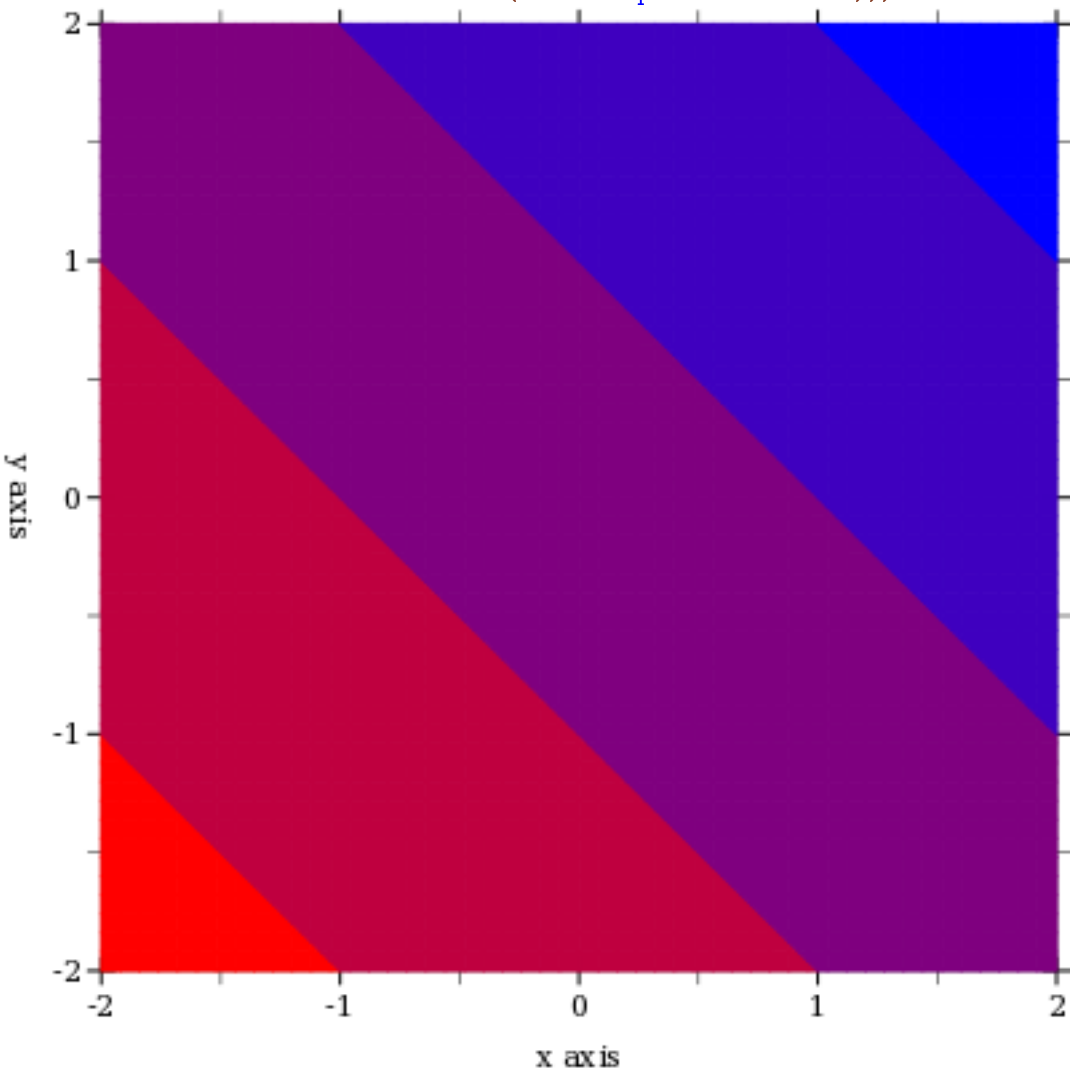
```
> (bounds->intervals (linear-seq 0 1 5))
(list  (ivl  0 1/4) (ivl  1/4 1/2) (ivl  1/2 3/4) (ivl  3/4 1))
```

```
(color-seq  c1
            c2
            num
           [#:start? start?
            #:end? end?])
 → (listof (list/c real? real? real?))
  c1 : color/c
  c2 : color/c
  num : exact-nonnegative-integer?
  start? : boolean? = #t
  end? : boolean? = #t
```

Interpolates between colors—red, green and blue components separately—using linear-seq. The #:start? and #:end? keyword arguments work as in linear-seq.

Example:

```
> (plot (contour-intervals (λ (x y) (+ x y)) -2 2 -2 2
                           #:levels 4 #:contour-
styles '(transparent)
                           #:colors (color-seq "red" "blue" 5)))
```



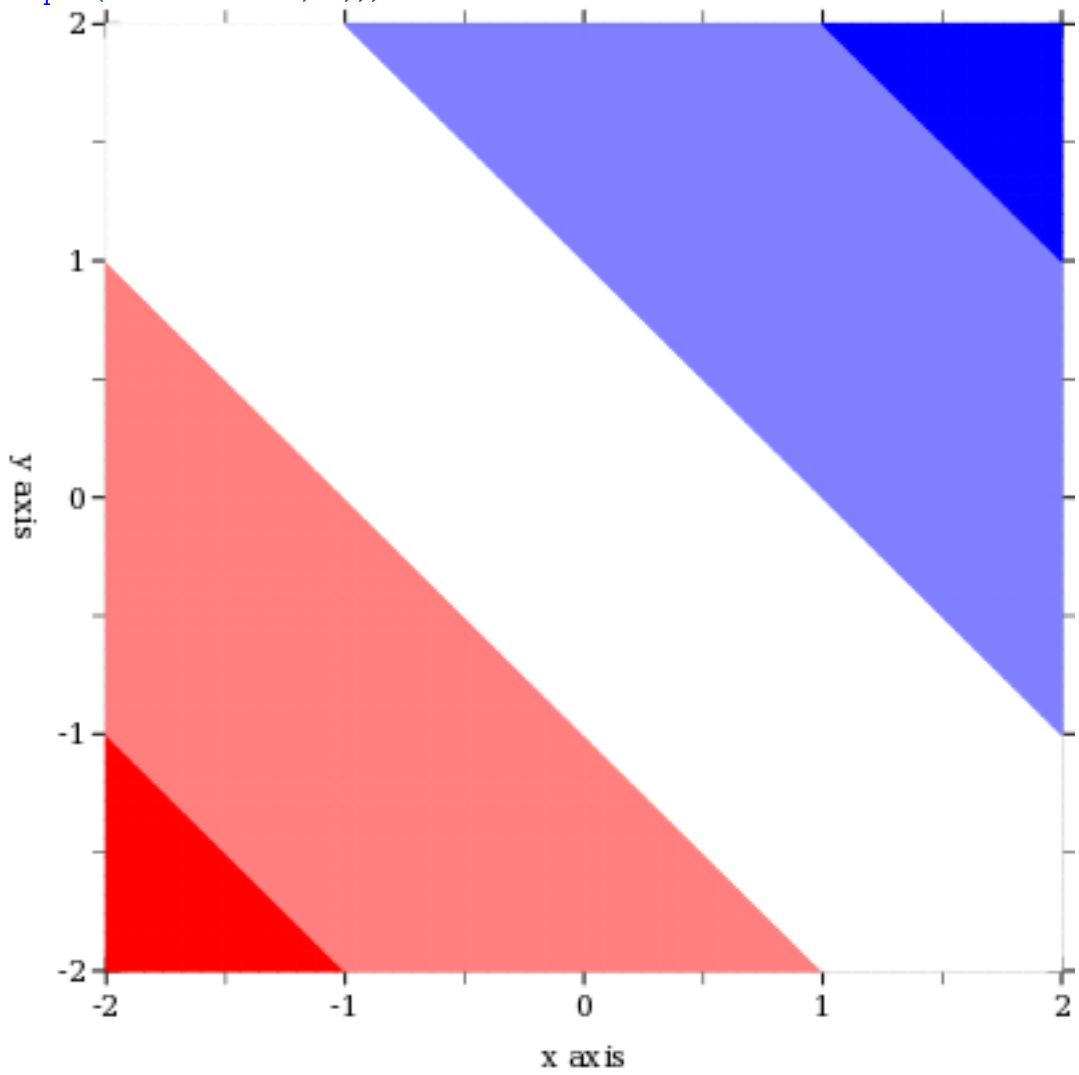

```
(color-seq* colors
            num
            [#:start? start?
             #:end? end?])
 → (listof (list/c real? real? real?))
  colors : (listof color/c)
  num : exact-nonnegative-integer?
  start? : boolean? = #t
```

```
end? : boolean? = #t
```

Interpolates between colors—red, green and blue components separately—using `linear-seq*`. The `#:start?` and `#:end?` keyword arguments work as in `linear-seq`.

Example:

```
> (plot (contour-intervals (λ (x y) (+ x y)) -2 2 -2 2
                           #:levels 4 #:contour-
styles '(transparent)
                           #:colors (color-
seq* '(red white blue) 5)))
```



```
(->color c) → (list/c real? real? real?)
  c : color/c
```

Converts a non-integer plot color to an RGB triplet.

Symbols are converted to strings, and strings are looked up in a `color-database<%>`. Lists are unchanged, and `color%` objects are converted straightforwardly.

Examples:

```
> (->color 'navy)
'(36 36 140)
> (->color "navy")
'(36 36 140)
> (->color '(36 36 140))
'(36 36 140)
> (->color (make-object color% 36 36 140))
'(36 36 140)
```

This function does not convert integers to RGB triplets, because there is no way for it to know whether the color will be used for a pen or for a brush. Use `->pen-color` and `->brush-color` to convert integers.

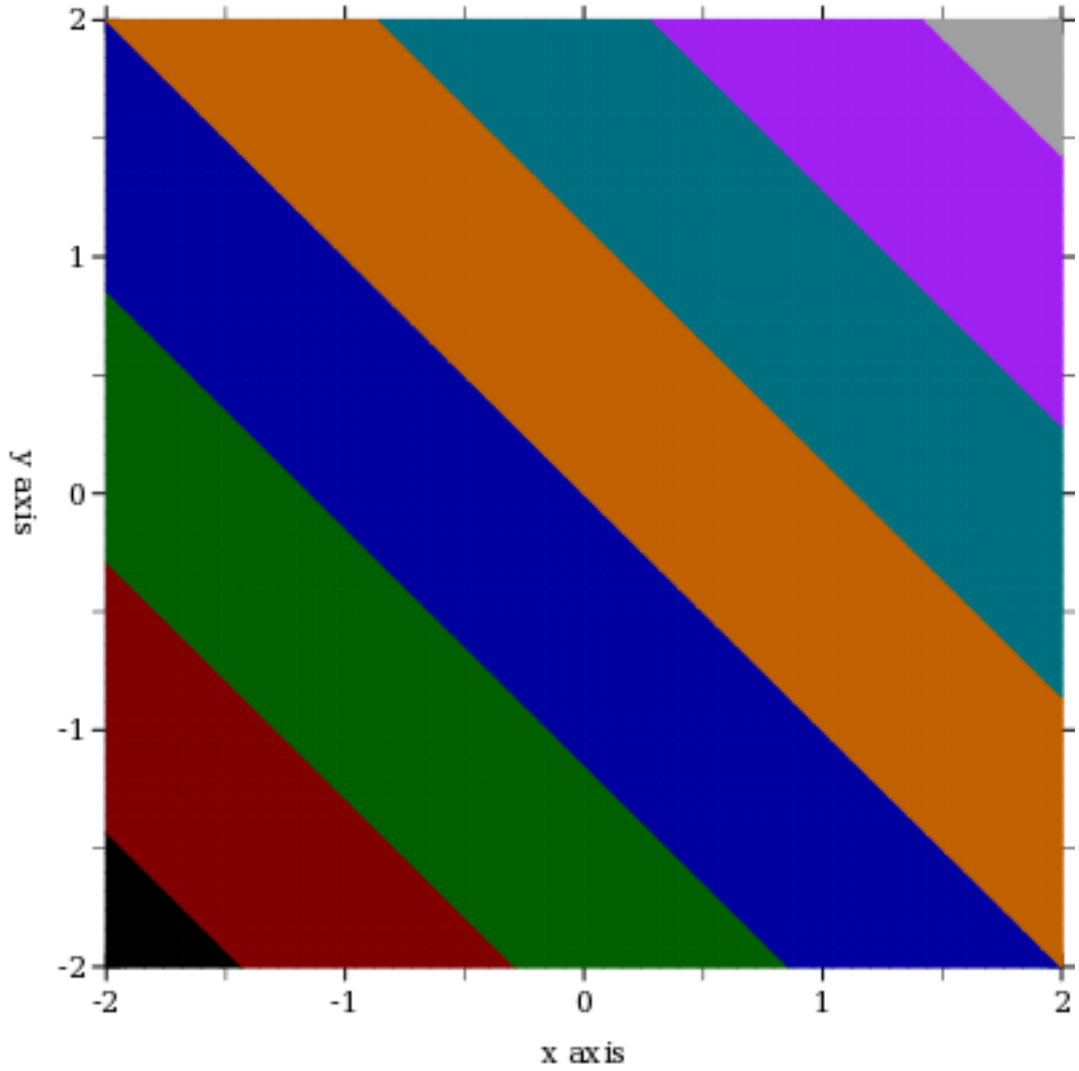
```
(->pen-color c) → (list/c real? real? real?)
  c : plot-color/c
```

Converts a *line* color to an RGB triplet. This function interprets integer colors as darker and more saturated than `->brush-color` does.

Non-integer colors are converted using `->color`. Integer colors are chosen for good pairwise contrast, especially between neighbors. Integer colors repeat starting with `8`.

Examples:

```
> (equal? (->pen-color 0) (->pen-color 8))
#t
> (plot (contour-intervals
         (λ (x y) (+ x y)) -2 2 -2 2
         #:levels 7 #:contour-styles '(transparent)
         #:colors (map ->pen-color (build-list 8 values))))
```




```
(->brush-color c) → (list/c real? real? real?)
  c : plot-color/c
```

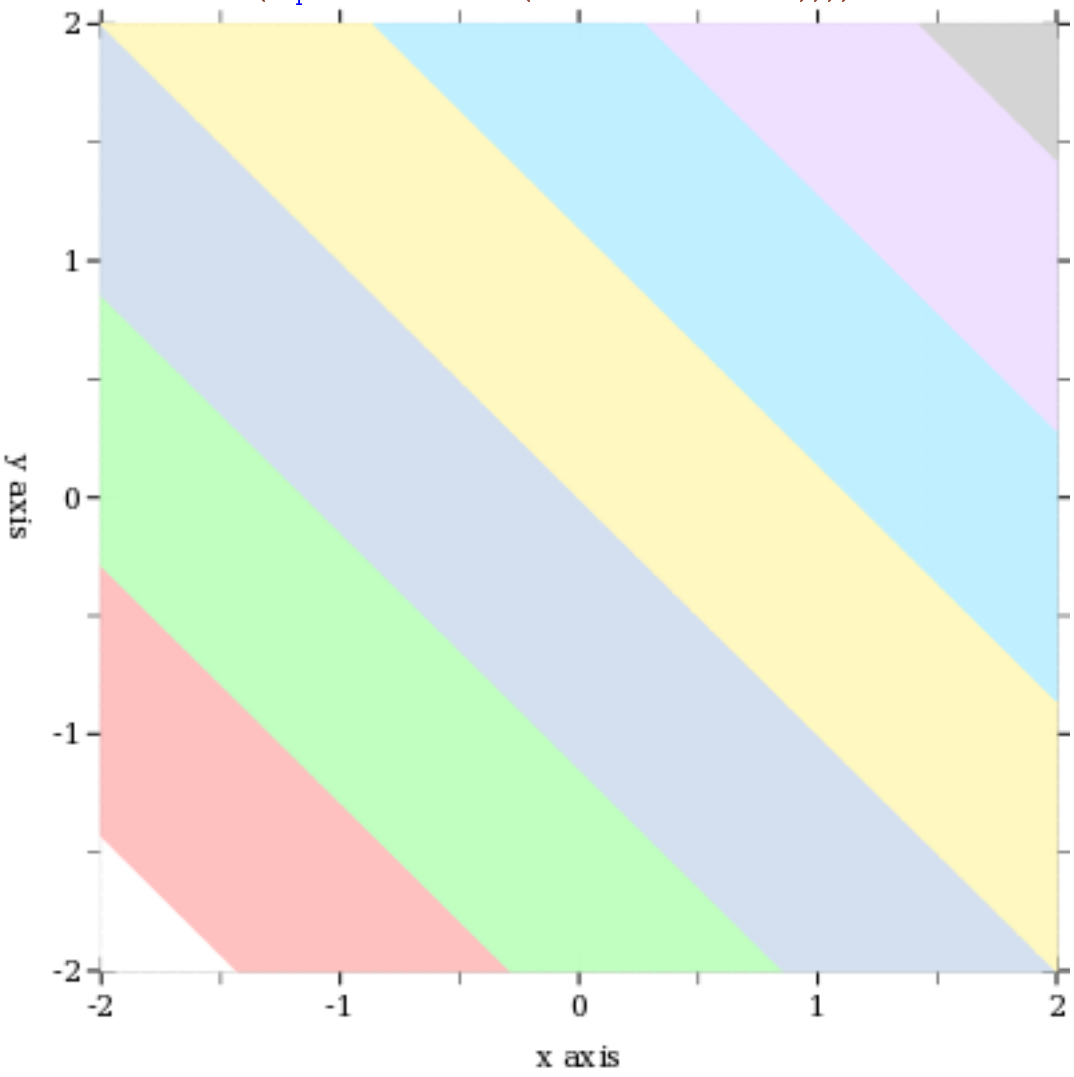
Converts a *fill* color to an RGB triplet. This function interprets integer colors as lighter and less saturated than `->pen-color` does.

Non-integer colors are converted using `->color`. Integer colors are chosen for good pairwise contrast, especially between neighbors. Integer colors repeat starting with `8`.

Examples:

```
> (equal? (->brush-color 0) (->brush-color 8))
#t
```

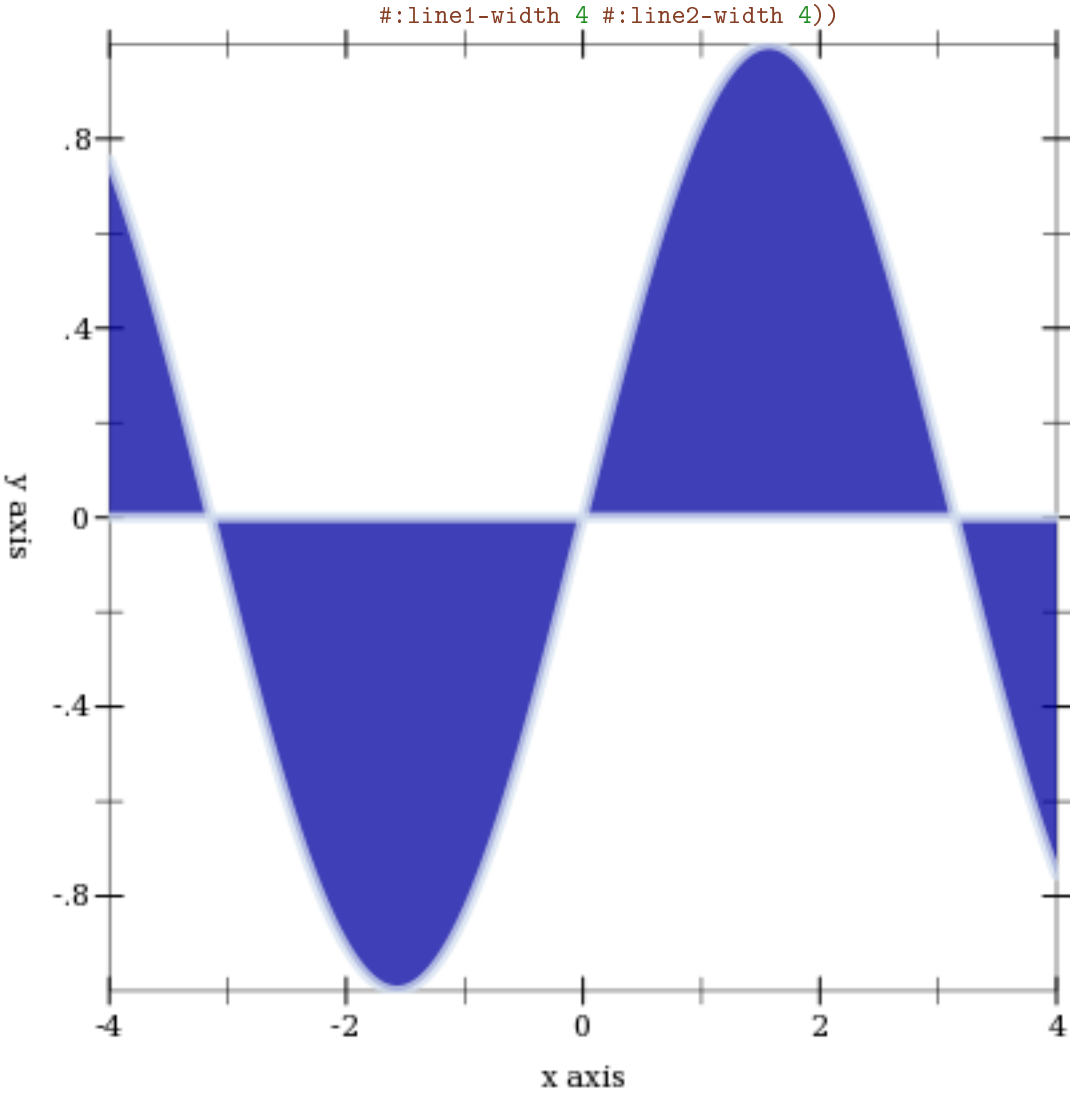
```
> (plot (contour-intervals
         (λ (x y) (+ x y)) -2 2 -2 2
         #:levels 7 #:contour-styles '(transparent)
         #:colors (map ->brush-color (build-list 8 values))))
```



In the above example, `map`ping `->brush-color` over the list is actually unnecessary, because `contour-intervals` uses `->brush-color` internally to convert fill colors.

The `function-interval` function generally plots areas using a fill color and lines using a line color. Both kinds of color have the default value `3`. The following example reverses the default behavior; i.e it draws areas using *line* color `3` and lines using *fill* color `3`:

```
> (plot (function-interval sin (λ (x) 0) -4 4
                           #:color (->pen-color 3)
                           #:line1-color (->brush-color 3)
                           #:line2-color (->brush-color 3)
                           #:line1-width 4 #:line2-width 4))
```




```
(->pen-style s) → symbol?
  s : plot-pen-style/c
```

Converts a symbolic pen style or a number to a symbolic pen style. Symbols are unchanged. Integer pen styles repeat starting at 5.

Examples:

```
> (eq? (->pen-style 0) (->pen-style 5))
#t
> (map ->pen-style '(0 1 2 3 4))
'(solid dot long-dash short-dash dot-dash)
```

```
(->brush-style s) → symbol?
  s : plot-brush-style/c
```

Converts a symbolic brush style or a number to a symbolic brush style. Symbols are unchanged. Integer brush styles repeat starting at `7`.

Examples:

```
> (eq? (->brush-style 0) (->brush-style 7))
#t
> (map ->brush-style '(0 1 2 3))
'(solid bdiagonal-hatch fdiagonal-hatch crossdiag-hatch)
> (map ->brush-style '(4 5 6))
'(horizontal-hatch vertical-hatch cross-hatch)
```

```
(struct invertible-function (f finv)
        #:extra-constructor-name make-invertible-function)
  f : (real? . -> . real?)
  finv : (real? . -> . real?)
```

Represents an invertible function.

The function itself is `f`, and its inverse is `finv`. Because `real?`s can be inexact, this invariant must be approximate and therefore cannot be enforced. (For example, `(exp (log 10))` = `10.000000000000002`.) The obligation to maintain it rests on whomever constructs one.

An axis transform such as `plot-x-transform` is a function from bounds `start end` to an `invertible-function` for which `(f start)` = `start` and `(f end)` = `end` (approximately), and the same is true of `finv`. The function `f` is used to transform points before drawing; its inverse `finv` is used to generate samples that will be evenly spaced after being transformed by `f`.

(Technically, because of the way PLoT uses `invertible-function`, `f` must only be a left inverse of `finv`; there is no requirement that `f` also be a right inverse of `finv`.)

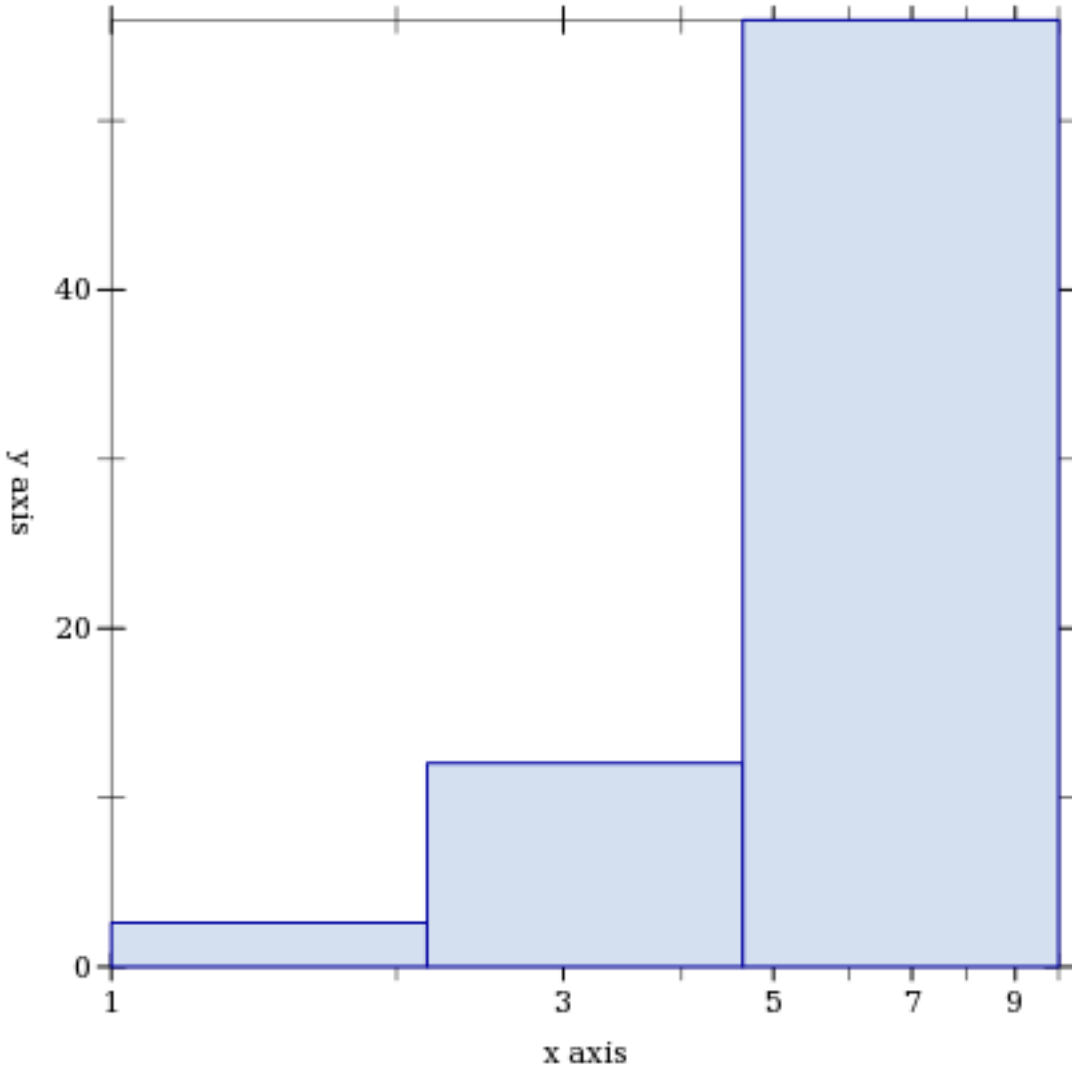
```
(nonlinear-seq  start
                end
                num
                transform
               [#:start? start?
                #:end? end?])    → (listof real?)
```

```
start : real?
end : real?
num : exact-nonnegative-integer?
transform : (real? real? . -> . invertible-function?)
start? : boolean? = #t
end? : boolean? = #t
```

Generates a list of reals that, if transformed using *transform*, would be evenly spaced. This is used to generate samples for transformed axes.

Examples:

```
> (linear-seq 1 10 4)
'(1 4 7 10)
> (nonlinear-seq 1 10 4 log-transform)
'(1.0 2.154434690031884 4.641588833612779 10.000000000000002)
> (parameterize ([plot-x-transform  log-transform])
    (plot (area-histogram sqr (nonlinear-seq 1 10 4 log-
transform))))
```




```
(struct mapped-function (f fmap)
        #:extra-constructor-name make-mapped-function)
  f : (any/c . -> . any/c)
  fmap : ((listof any/c) . -> . (listof any/c))
```

Represents a function that maps over lists differently than `(map f xs)`.

With some functions, mapping over a list can be done much more quickly if done specially. (An example is a piecewise function with many pieces that first must decide which interval its input belongs to. Deciding that for many inputs can be done more efficiently by sorting all the inputs first.) Renderer-producing functions that accept a `(-> real? real?)` also accept a `mapped-function`, and use its `fmap` to sample more efficiently.

```
                mapped-function?
(kde xs h) → (or/c real? #f)
                (or/c real? #f)
  xs : (listof real?)
  h : real?
```

Given samples and a kernel bandwidth, returns a `mapped-function` representing a kernel density estimate, and bounds, outside of which the density estimate is zero. Used by `density`.

7 Plot and Renderer Parameters

7.1 Compatibility

```
(plot-deprecation-warnings?) → boolean?
(plot-deprecation-warnings? warnings?) → void?
  warnings? : boolean?
= #f
```

When `#t`, prints a deprecation warning to `current-error-port` on the first use of `mix`, `line`, `contour`, `shade`, `surface`, or a keyword argument of `plot` or `plot3d` that exists solely for backward compatibility.

7.2 Output

```
(plot-new-window?) → boolean?
(plot-new-window? window?) → void?
  window? : boolean?
= #f
```

When `#t`, `plot` and `plot3d` open a new window for each plot instead of returning an `image-snip%`.

Users of command-line Racket, which cannot display image snips, should enter

```
(plot-new-window? #t)
```

before using `plot` or `plot3d`.

```
(plot-width) → exact-positive-integer?
(plot-width width) → void?
  width : exact-positive-integer?
= 400
```

```
(plot-height) → exact-positive-integer?
(plot-height height) → void?
  height : exact-positive-integer?
```

```
= 400
```

The width and height of a plot, in logical drawing units (e.g. pixels for bitmap plots).

```
(plot-jpeg-quality) → (integer-in 0 100)
(plot-jpeg-quality quality) → void?
  quality : (integer-in 0 100)
= 100
```

The quality of JPEG images written by `plot-file` and `plot3d-file`. See `save-file`.

```
(plot-ps/pdf-interactive?) → boolean?
(plot-ps/pdf-interactive? interactive?) → void?
  interactive? : boolean?
= #f
```

If `#t`, `plot-file` and `plot3d-file` open a dialog when writing PostScript or PDF files. See `post-script-dc%` and `pdf-dc%`.

7.3 Axis Transforms

```
(plot-x-transform) → (real? real? . -> . invertible-function?)
(plot-x-transform transform) → void?
  transform : (real? real? . -> . invertible-function?)
= id-transform
```

```
(plot-y-transform) → (real? real? . -> . invertible-function?)
(plot-y-transform transform) → void?
  transform : (real? real? . -> . invertible-function?)
= id-transform
```

```
(plot-z-transform) → (real? real? . -> . invertible-function?)
(plot-z-transform transform) → void?
  transform : (real? real? . -> . invertible-function?)
```

```
= id-transform
```

Per-axis, nonlinear transforms. Set these, for example, to plot with log-scale axes. See log-transform.

```
(id-transform x-min x-max) → invertible-function?
  x-min : real?
  x-max : real?
```

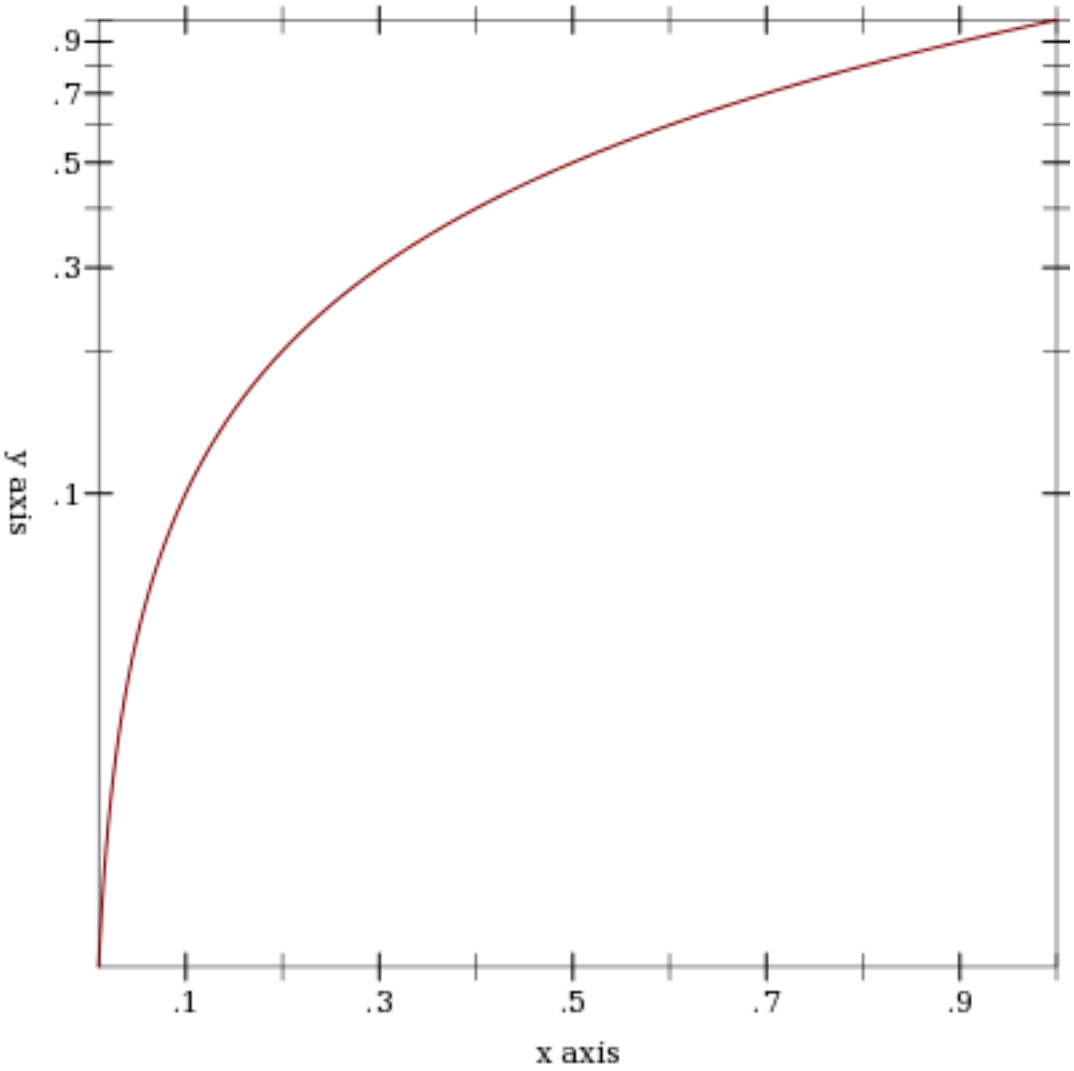
The default transform for all axes.

```
(log-transform x-min x-max) → invertible-function?
  x-min : real?
  x-max : real?
```

A log transform. Use this to generate plots with log-scale axes. Any log-scaled axis must be on a positive interval.

Examples:

```
> (parameterize ([plot-y-transform  log-transform])
    (plot (function (λ (x) x) 0.01 1)))
```




```
> (parameterize ([plot-x-transform  log-transform])
    (plot (function (λ (x) x) -1 1)))
log-transform: expects type <positive real> as 1st
argument, given: -1; other arguments were: 1
```

```
(cbrt-transform x-min x-max) → invertible-function?
  x-min : real?
  x-max : real?
```

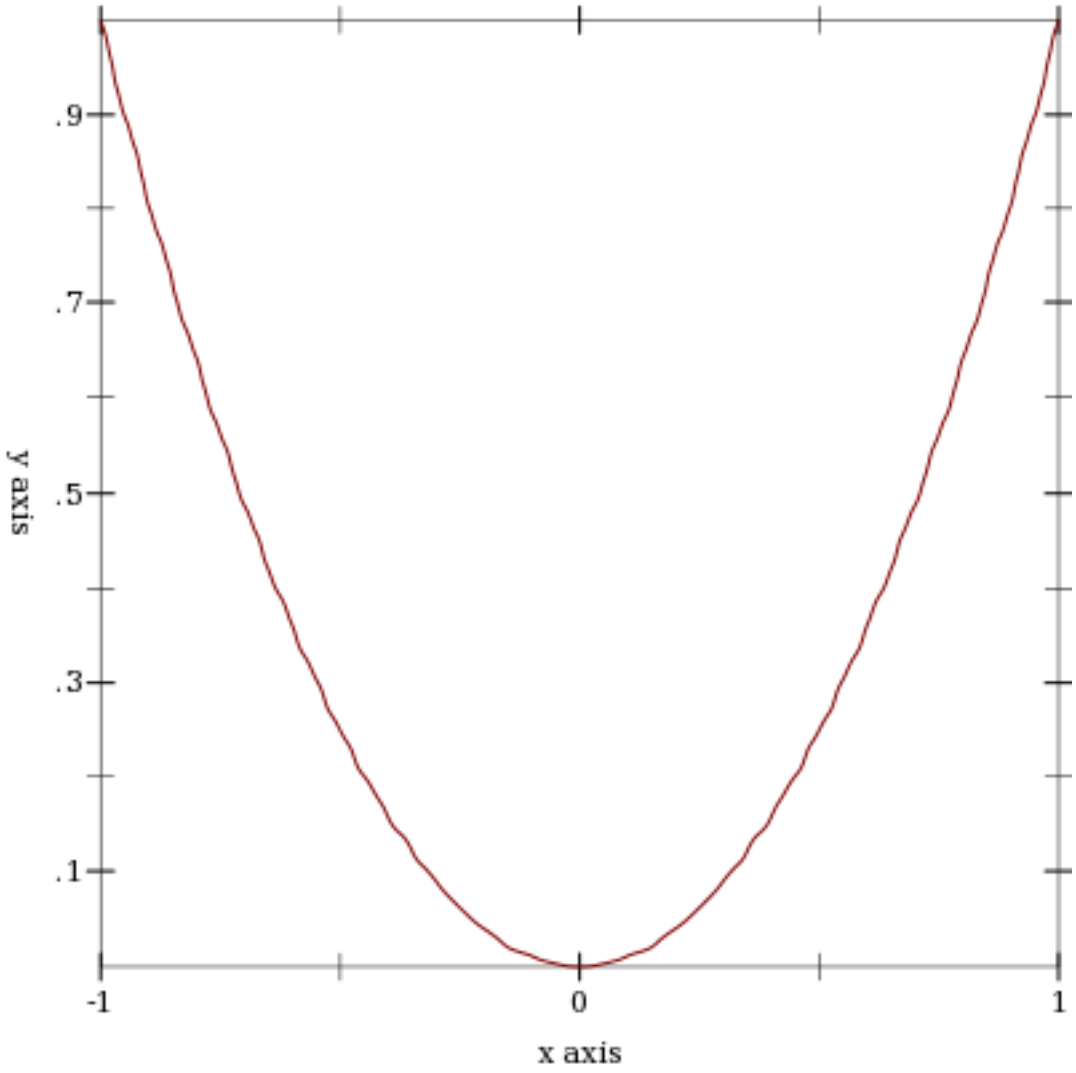
A "cube-root" transform. Unlike the log transform, it is defined on the entire real line, making it better for testing the appearance of plots with nonlinearly transformed axes.

```
(hand-drawn-transform freq)
 → (real? real? . -> . invertible-function?)
  freq : (>/c 0)
```

An *extremely important* test case, which makes sure that PLoT can use any monotone, invertible function as an axis transform.

The *freq* parameter controls the "shakiness" of the transform. At high values, it makes plots look like Peanuts cartoons. For example,

```
> (parameterize ([plot-x-transform  (hand-drawn-transform 200)]
                 [plot-y-transform  (hand-drawn-transform 200)])
    (plot (function sqr -1 1)))
```



7.4 General Appearance

```
(plot-foreground) → plot-color/c
(plot-foreground color) → void?
  color : plot-color/c
```

```
= 0
```

```
(plot-background) → plot-color/c
(plot-background color) → void?
  color : plot-color/c
```

```
= 0
```

The plot foreground and background color. That both are `0` by default is not a mistake: for foreground colors, `0` is interpreted as black; for background colors, `0` is interpreted as white. See `->pen-color` and `->brush-color` for details on how PLoT interprets integer colors.

```
(plot-foreground-alpha) → (real-in 0 1)
(plot-foreground-alpha alpha) → void?
  alpha : (real-in 0 1)
```

```
= 1
```

```
(plot-background-alpha) → (real-in 0 1)
(plot-background-alpha alpha) → void?
  alpha : (real-in 0 1)
```

```
= 1
```

The opacity of the background and foreground colors.

```
(plot-font-size) → (>=/c 0)
(plot-font-size size) → void?
  size : (>=/c 0)
```

```
= 11
```

The font size of the title, axis labels, tick labels, and other labels, in drawing units.

```
(plot-font-family) → font-family/c
(plot-font-family family) → void?
  family : font-family/c
```

```
= 'roman
```

The font family used for the title and labels.

```
(plot-line-width) → (>=/c 0)
(plot-line-width width) → void?
  width : (>=/c 0)
= 1
```

The width of axis lines and major tick lines. (Minor tick lines are half this width.)

```
(plot-legend-anchor) → anchor/c
(plot-legend-anchor anchor) → void?
  anchor : anchor/c
= 'top-right
```

Controls the placement of the legend.

```
(plot-legend-box-alpha) → (real-in 0 1)
(plot-legend-box-alpha alpha) → void?
  alpha : (real-in 0 1)
= 2/3
```

The opacity of the filled rectangle behind the legend entries.

```
(plot-tick-size) → (>=/c 0)
(plot-tick-size size) → void?
  size : (>=/c 0)
= 10
```

The length of tick lines, in drawing units.

```
(plot-title) → (or/c string? #f)
(plot-title title) → void?
  title : (or/c string? #f)
= #f
```

```
(plot-x-label) → (or/c string? #f)
(plot-x-label label) → void?
  label : (or/c string? #f)
= "x axis"
```

```
(plot-y-label) → (or/c string? #f)
(plot-y-label label) → void?
  label : (or/c string? #f)
= "y axis"
```

```
(plot-z-label) → (or/c string? #f)
(plot-z-label label) → void?
  label : (or/c string? #f)
= #f
```

The title and axis labels. A `#f` value means the label is not drawn and takes no space. A `""` value effectively means the label is not drawn, but it takes space.

```
(plot-animating?) → boolean?
(plot-animating? animating?) → void?
  animating? : boolean?
= #f
```

When `#t`, certain renderers draw simplified plots to speed up drawing. PLoT sets it to `#t`, for example, when a user is clicking and dragging a 3D plot to rotate it.

7.5 Lines

```
(line-samples) → (and/c exact-integer? (>=/c 2))
(line-samples samples) → void?
  samples : (and/c exact-integer? (>=/c 2))
= 500
```

```
(line-color) → plot-color/c
(line-color color) → void?
  color : plot-color/c
= 1
```

```
(line-width) → (>=/c 0)
(line-width width) → void?
  width : (>=/c 0)
= 1
```

```
(line-style) → plot-pen-style/c
(line-style style) → void?
  style : plot-pen-style/c
= 'solid
```

```
(line-alpha) → (real-in 0 1)
(line-alpha alpha) → void?
  alpha : (real-in 0 1)
= 1
```

7.6 Intervals

```
(interval-color) → plot-color/c
(interval-color color) → void?
  color : plot-color/c
= 3
```

```
(interval-style) → plot-brush-style/c
(interval-style style) → void?
  style : plot-brush-style/c
```

```
= 'solid
```

```
(interval-line1-color) → plot-color/c
(interval-line1-color color) → void?
  color : plot-color/c
```

```
= 3
```

```
(interval-line1-width) → (>=/c 0)
(interval-line1-width width) → void?
  width : (>=/c 0)
```

```
= 1
```

```
(interval-line1-style) → plot-pen-style/c
(interval-line1-style style) → void?
  style : plot-pen-style/c
```

```
= 'solid
```

```
(interval-line2-color) → plot-color/c
(interval-line2-color color) → void?
  color : plot-color/c
```

```
= 3
```

```
(interval-line2-width) → (>=/c 0)
(interval-line2-width width) → void?
  width : (>=/c 0)
```

```
= 1
```

```
(interval-line2-style) → plot-pen-style/c
(interval-line2-style style) → void?
  style : plot-pen-style/c
```

```
= 'solid
```

```
(interval-alpha) → (real-in 0 1)
(interval-alpha alpha) → void?
  alpha : (real-in 0 1)
```

```
= 3/4
```

7.7 Points

```
(point-sym) → point-sym/c
(point-sym sym) → void?
  sym : point-sym/c
```

```
= 'circle
```

```
(point-color) → plot-color/c
(point-color color) → void?
  color : plot-color/c
```

```
= 0
```

```
(point-size) → (>=/c 0)
(point-size size) → void?
  size : (>=/c 0)
```

```
= 6
```

```
(point-line-width) → (>=/c 0)
(point-line-width width) → void?
  width : (>=/c 0)
```

```
= 1
```

```
(point-alpha) → (real-in 0 1)
(point-alpha alpha) → void?
  alpha : (real-in 0 1)
```

```
= 1
```

7.8 Vector Fields

```
(vector-field-samples) → exact-positive-integer?
(vector-field-samples samples) → void?
  samples : exact-positive-integer?
```

```
= 20
```

```
(vector-field-color) → plot-color/c
(vector-field-color color) → void?
  color : plot-color/c
```

```
= 1
```

```
(vector-field-line-width) → (>=/c 0)
(vector-field-line-width width) → void?
  width : (>=/c 0)
```

```
= 2/3
```

```
(vector-field-line-style) → plot-pen-style/c
(vector-field-line-style style) → void?
  style : plot-pen-style/c
```

```
= 'solid
```

```
(vector-field-scale)
 → (or/c real? (one-of/c 'auto 'normalized))
(vector-field-scale scale) → void?
  scale : (or/c real? (one-of/c 'auto 'normalized))
```

```
= 'auto
```

```
(vector-field-alpha) → (real-in 0 1)
(vector-field-alpha alpha) → void?
  alpha : (real-in 0 1)
```

```
= 1
```

7.9 Error Bars

```
(error-bar-width) → (>=/c 0)
(error-bar-width width) → void?
  width : (>=/c 0)
```

```
= 6
```

```
(error-bar-color) → plot-color/c
(error-bar-color color) → void?
  color : plot-color/c
```

```
= 0
```

```
(error-bar-line-width) → (>=/c 0)
(error-bar-line-width width) → void?
  width : (>=/c 0)
```

```
= 1
```

```
(error-bar-line-style) → plot-pen-style/c
(error-bar-line-style style) → void?
  style : plot-pen-style/c
= 'solid
```

```
(error-bar-alpha) → (real-in 0 1)
(error-bar-alpha alpha) → void?
  alpha : (real-in 0 1)
= 2/3
```

7.10 Contours and Contour Intervals

```
(default-contour-colors zs) → (listof plot-color/c)
  zs : (listof real?)
```

```
(default-contour-fill-colors zs) → (listof plot-color/c)
  zs : (listof real?)
```

```
(contour-samples) → (and/c exact-integer? (>=/c 2))
(contour-samples samples) → void?
  samples : (and/c exact-integer? (>=/c 2))
= 51
```

```
(contour-levels)
 → (or/c 'auto exact-positive-integer? (listof real?))
(contour-levels levels) → void?
  levels : (or/c 'auto exact-positive-integer? (listof real?))
= 'auto
```

```
(contour-colors) → plot-colors/c
(contour-colors colors) → void?
  colors : plot-colors/c
```

```
= default-contour-colors
```

```
(contour-widths) → pen-widths/c
(contour-widths widths) → void?
  widths : pen-widths/c
= '(1)
```

```
(contour-styles) → plot-pen-styles/c
(contour-styles styles) → void?
  styles : plot-pen-styles/c
= '(solid long-dash)
```

```
(contour-alphas) → alphas/c
(contour-alphas alphas) → void?
  alphas : alphas/c
= '(1)
```

```
(contour-interval-colors) → plot-colors/c
(contour-interval-colors colors) → void?
  colors : plot-colors/c
= default-contour-fill-colors
```

```
(contour-interval-styles) → plot-brush-styles/c
(contour-interval-styles styles) → void?
  styles : plot-brush-styles/c
= '(solid)
```

```
(contour-interval-alphas) → alphas/c
(contour-interval-alphas alphas) → void?
  alphas : alphas/c
```

```
= '(1)
```

7.11 Rectangles

```
(rectangle-color) → plot-color/c
(rectangle-color color) → void?
  color : plot-color/c
= 3
```

```
(rectangle-style) → plot-brush-style/c
(rectangle-style style) → void?
  style : plot-brush-style/c
= 'solid
```

```
(rectangle-line-color) → plot-color/c
(rectangle-line-color color) → void?
  color : plot-color/c
= 3
```

```
(rectangle-line-width) → (>=/c 0)
(rectangle-line-width width) → void?
  width : (>=/c 0)
= 1
```

```
(rectangle-line-style) → plot-pen-style/c
(rectangle-line-style style) → void?
  style : plot-pen-style/c
= 'solid
```

```
(rectangle-alpha) → (real-in 0 1)
(rectangle-alpha alpha) → void?
  alpha : (real-in 0 1)
= 1
```

```
(rectangle3d-line-width) → (>=/c 0)
(rectangle3d-line-width width) → void?
  width : (>=/c 0)
= 1/3
```

```
(discrete-histogram-gap) → (real-in 0 1)
(discrete-histogram-gap gap) → void?
  gap : (real-in 0 1)
= 1/8
```

7.12 Decorations

These parameters do not control the *typical* appearance of plots. Instead, they control the look of renderers that add specific decorations, such as labeled points.

```
(x-axis-ticks?) → boolean?
(x-axis-ticks? ticks?) → void?
  ticks? : boolean?
= #t
```

```
(y-axis-ticks?) → boolean?
(y-axis-ticks? ticks?) → void?
  ticks? : boolean?
= #t
```

```
(z-axis-ticks?) → boolean?
(z-axis-ticks? ticks?) → void?
  ticks? : boolean?
= #t
```

```
(polar-axes-number) → exact-positive-integer?
(polar-axes-number number) → void?
  number : exact-positive-integer?
= 12
```

```
(polar-axes-ticks?) → boolean?
(polar-axes-ticks? ticks?) → void?
  ticks? : boolean?
= #t
```

```
(label-anchor) → anchor/c
(label-anchor anchor) → void?
  anchor : anchor/c
= 'left
```

```
(label-angle) → real?
(label-angle angle) → void?
  angle : real?
= 0
```

```
(label-alpha) → (real-in 0 1)
(label-alpha alpha) → void?
  alpha : (real-in 0 1)
= 1
```

```
(label-point-size) → (>=/c 0)
(label-point-size size) → void?
  size : (>=/c 0)
= 4
```

7.13 3D General Appearance

```
(plot3d-samples) → (and/c exact-integer? (>=/c 2))
(plot3d-samples samples) → void?
  samples : (and/c exact-integer? (>=/c 2))
= 41
```

```
(plot3d-angle) → real?
(plot3d-angle angle) → void?
  angle : real?
= 30
```

```
(plot3d-altitude) → real?
(plot3d-altitude altitude) → void?
  altitude : real?
= 60
```

```
(plot3d-ambient-light) → (real-in 0 1)
(plot3d-ambient-light light) → void?
  light : (real-in 0 1)
= 2/3
```

```
(plot3d-diffuse-light?) → boolean?
(plot3d-diffuse-light? light?) → void?
  light? : boolean?
```

```
= #t
```

```
(plot3d-specular-light?) → boolean?
(plot3d-specular-light? light?) → void?
  light? : boolean?
```

```
= #t
```

7.14 Surfaces

```
(surface-color) → plot-color/c
(surface-color color) → void?
  color : plot-color/c
```

```
= 0
```

```
(surface-style) → plot-brush-style/c
(surface-style style) → void?
  style : plot-brush-style/c
```

```
= 'solid
```

```
(surface-line-color) → plot-color/c
(surface-line-color color) → void?
  color : plot-color/c
```

```
= 0
```

```
(surface-line-width) → (>=/c 0)
(surface-line-width width) → void?
  width : (>=/c 0)
```

```
= 1/3
```

```
(surface-line-style) → plot-pen-style/c
(surface-line-style style) → void?
  style : plot-pen-style/c
```

```
= 'solid
```

```
(surface-alpha) → (real-in 0 1)
(surface-alpha alpha) → void?
  alpha : (real-in 0 1)
```

```
= 1
```

7.15 Contour Surfaces

Contour surface renderers use shared contour parameters except for the following three.

```
(contour-interval-line-colors) → plot-colors/c
(contour-interval-line-colors colors) → void?
  colors : plot-colors/c
```

```
= '(0)
```

```
(contour-interval-line-widths) → pen-widths/c
(contour-interval-line-widths widths) → void?
  widths : pen-widths/c
```

```
= '(1/3)
```

```
(contour-interval-line-styles) → plot-pen-styles/c
(contour-interval-line-styles styles) → void?
  styles : plot-pen-styles/c
```

```
= '(solid)
```

7.16 Isosurfaces

Single isosurfaces (isosurface3d) use surface parameters. Nested isosurfaces (isosurfaces3d) use the following.

```
(default-isosurface-colors zs) → (listof plot-color/c)
  zs : (listof real?)
```

```
(default-isosurface-line-colors zs) → (listof plot-color/c)
  zs : (listof real?)
```

```
(isosurface-levels) → exact-positive-integer?
(isosurface-levels levels) → void?
  levels : exact-positive-integer?
= 3
```

```
(isosurface-colors) → plot-colors/c
(isosurface-colors colors) → void?
  colors : plot-colors/c
= default-isosurface-colors
```

```
(isosurface-line-colors) → plot-colors/c
(isosurface-line-colors colors) → void?
  colors : plot-colors/c
= default-isosurface-line-colors
```

```
(isosurface-line-widths) → pen-widths/c
(isosurface-line-widths widths) → void?
  widths : pen-widths/c
= '(1/3)
```

```
(isosurface-line-styles) → plot-pen-styles/c
(isosurface-line-styles styles) → void?
  styles : plot-pen-styles/c
```

```
= '(solid)
```

```
(isosurface-alphas) → alphas/c
(isosurface-alphas alphas) → void?
  alphas : alphas/c
= '(1/2)
```

8 Plot Contracts

8.1 Convenience Contracts

```
(treeof ct) → contract?
  ct : (or/c contract? (any/c . -> . any/c))
= (or/c ct (listof (recursive-contract (treeof ct))))
```

Identifies trees of values that meet the contract `ct`. Used by `plot` and `plot3d` to construct the contract for a tree of `renderer2d?` or `renderer3d?`.

8.2 Appearance Argument Contracts

```
anchor/c : contract?
= (one-of/c 'top-left 'top 'top-right
            'left 'center 'right
            'bottom-left 'bottom 'bottom-right)
```

The contract for `anchor` arguments and parameters, such as `plot-legend-anchor`.

```
color/c : contract?
= (or/c (list/c real? real? real?)
        string? symbol?
        (is-a?/c color%))
```

```
plot-color/c : contract?
= (or/c exact-integer? color/c)
```

The contract for `#:color` arguments, and parameters such as `line-color` and `surface-color`. For the meaning of integer colors, see `->pen-color` and `->brush-color`.

```
plot-pen-style/c : contract?
```

```
= (or/c exact-integer?
        (one-of/c 'transparent 'solid 'dot 'long-dash
                  'short-dash 'dot-dash))
```

The contract for `#:style` arguments (when they refer to lines), and paramters such as `line-style`. For the meaning of integer pen styles, see `->pen-style`.

```
plot-brush-style/c : contract?
= (or/c exact-integer?
        (one-of/c 'transparent 'solid
                  'bdiagonal-hatch 'fdiagonal-hatch 'crossdiag-hatch
                  'horizontal-hatch 'vertical-hatch 'cross-hatch))
```

The contract for `#:style` arguments (when they refer to fills), and parameters such as `interval-style`. For the meaning of integer brush styles, see `->brush-style`.

```
font-family/c : contract?
= (one-of/c 'default 'decorative 'roman 'script 'swiss
            'modern 'symbol 'system)
```

Identifies legal font family values. See `plot-font-family`.

```
point-sym/c : contract?
= (or/c char? string? integer? (apply one-of/c known-point-symbols))
```

The contract for the `#:sym` arguments in `points` and `points3d`, and the parameter `point-sym`.

```
known-point-symbols : (listof symbol?)
```

A list containing the symbols that are valid `points` symbols.

```
> (require (only-in srfi/13 string-pad-right))
> (for ([sym  (in-list known-point-symbols)]
        [n    (in-cycle (in-range 3))])
    (display (string-pad-right (format "~v" sym) 22))
    (when (= n 2) (newline)))
'dot                   'point                 'pixel
'plus                  'times                 'asterisk
```

```
'5asterisk            'odot                 'oplus
'otimes               'oasterisk            'o5asterisk
'circle               'square               'diamond
'triangle             'fullcircle           'fullsquare
'fulldiamond          'fulltriangle         'triangleup
'triangledown         'triangleleft         'triangleright
'fulltriangleup       'fulltriangledown     'fulltriangleleft
'fulltriangleright    'rightarrow           'leftarrow
'uparrow              'downarrow            '4star
'5star                '6star                '7star
'8star                'full4star            'full5star
'full6star            'full7star            'full8star
'circle1              'circle2              'circle3
'circle4              'circle5              'circle6
'circle7              'circle8              'bullet
'fullcircle1          'fullcircle2          'fullcircle3
'fullcircle4          'fullcircle5          'fullcircle6
'fullcircle7          'fullcircle8
> (length known-point-symbols)
59
```

8.3 Appearance Argument Sequence Contracts

```
plot-colors/c : contract?

= (or/c (listof plot-color/c)
        ((listof real?) . -> . (listof plot-color/c)))
```

The contract for `#:colors` arguments, as in `contours`. If the contracted value is a function, it is intended to take a list of values, such as contour values, as input, and return a list of colors. The number of colors need not be the same as the number of values. If shorter, they will cycle; if longer, some will not be used.

See `color-seq` and `color-seq*` for a demonstration of constructing arguments with this contract.

```
plot-pen-styles/c : contract?

= (or/c (listof plot-pen-style/c)
        ((listof real?) . -> . (listof plot-pen-style/c)))
```

Like `plot-colors/c`, but for line styles.

```
pen-widths/c : contract?
= (or/c (listof (>=/c 0))
        ((listof real?) . -> . (listof (>=/c 0))))
```

Like plot-colors/c, but for line widths.

```
plot-brush-styles/c : contract?
= (or/c (listof plot-brush-style/c)
        ((listof real?) . -> . (listof plot-brush-style/c)))
```

Like plot-colors/c, but for fill styles.

```
alphas/c : contract?
= (or/c (listof (real-in 0 1))
        ((listof real?) . -> . (listof (real-in 0 1))))
```

Like plot-colors/c, but for opacities.

9 Making Custom Plot Renderers

```
(require plot/custom)
```

Eventually, enough of the underlying PLoT API will be exposed that anyone can create new renderers. However, the underlying API still changes too often. As soon as it settles, `plot/custom` will export it, and this page will document how to use it.

10 Porting From PLoT <= 5.1.3

If it seems porting will take too long, you can get your old code running more quickly using the §11 “Compatibility Module”.

The update from PLoT version 5.1.3 to 5.2 introduces a few incompatibilities:

- PLoT now allows plot elements to request plot area bounds, and finds bounds large enough to fit all plot elements. The old default plot area bounds of [-5,5] × [-5,5] cannot be made consistent with the improved behavior; the default bounds are now "no bounds". This causes code such as `(plot (line sin))`, which does not state bounds, to fail.
- The `#:width` and `#:style` keyword arguments to `vector-field` have been replaced by `#:line-width` and `#:scale` to be consistent with other functions.
- The `plot` function no longer takes a `(-> (is-a?/c 2d-view%) void?)` as an argument, but a `(treeof renderer2d?)`. The argument change in `plot3d` is similar. This should not affect most code because PLoT encourages regarding these data types as black boxes.
- The `plot-extend` module no longer exists.

This section of the PLoT manual will help you port code written for PLoT 5.1.3 and earlier to the most recent PLoT. There are four main tasks:

- Replace deprecated functions.
- Ensure that plots have bounds.
- Change `vector-field`, `plot` and `plot3d` keyword arguments.
- Fix broken calls to `points`.

You should also set `(plot-deprecation-warnings? #t)` to be alerted to uses of deprecated features.

10.1 Replacing Deprecated Functions

Replace `mix` with `list`, and replace `surface` with `surface3d`. These functions are drop-in replacements, but `surface3d` has many more features (and a name more consistent with similar functions).

Replace `line` with `function`, `parametric` or `polar`, depending on the keyword arguments to `line`. These are not at all drop-in replacements, but finding the right arguments should be straightforward.

Replace `contour` with `contours`, and replace `shade` with `contour-intervals`. These are *mostly* drop-in replacements: they should always work, but may not place contours at the same values (unless the levels are given as a list of values). For example, the default `#:levels` argument is now `'auto`, which chooses contour values in the same way that z axis tick locations are usually chosen in 3D plots. The number of contour levels is therefore some number between `4` and `10`, depending on the plot.

10.2 Ensuring That Plots Have Bounds

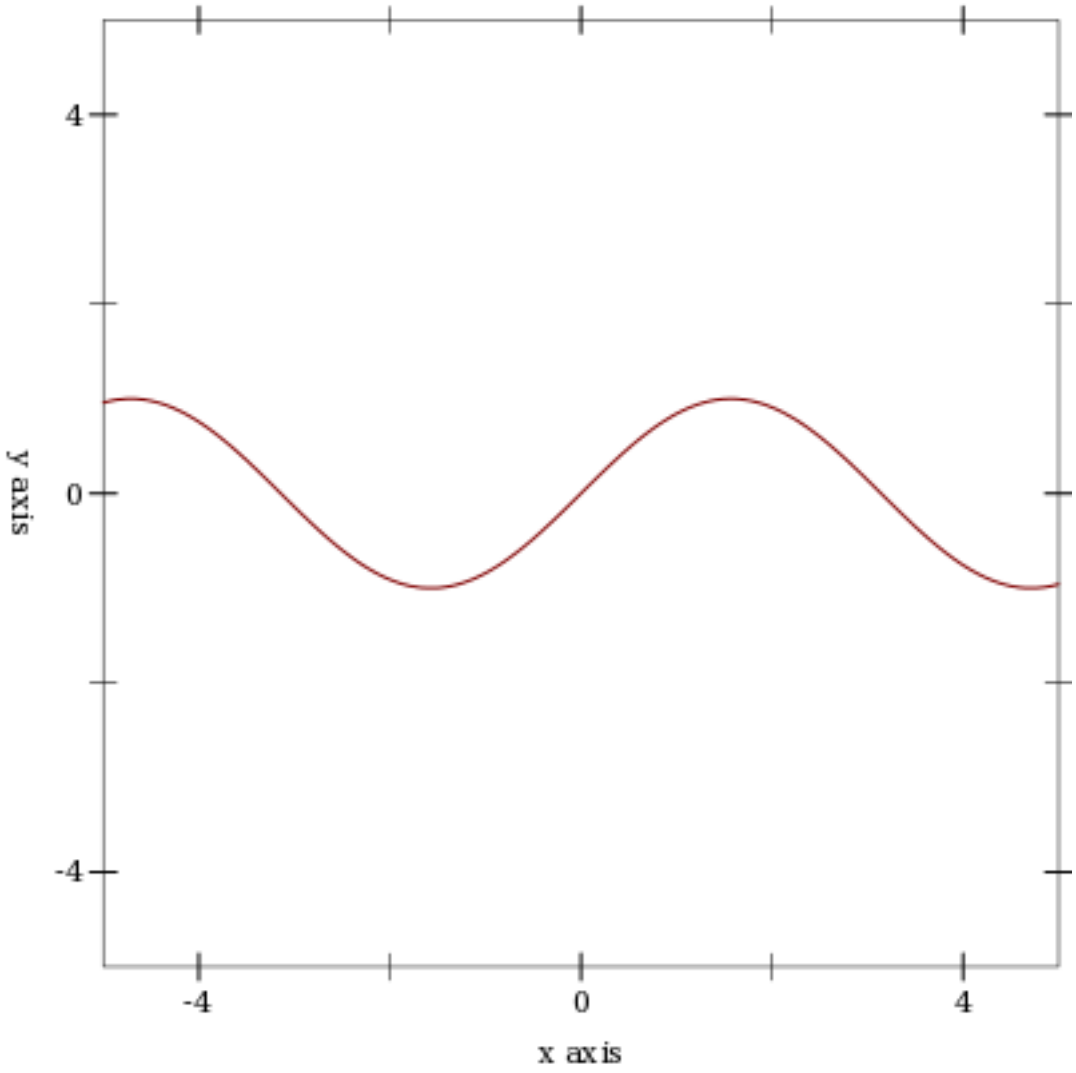
The safest way to ensure that `plot` can determine bounds for the plot area is to add `#:x-min -5 #:x-max 5 #:y-min -5 #:y-max 5` to every call to `plot`. Similarly, add `#:x-min -5 #:x-max 5 #:y-min -5 #:y-max 5 #:z-min -5 #:z-max 5` to every call to `plot3d`.

Because PLoT is now smarter about choosing bounds, there are better ways. For example, suppose you have

```
> (plot (line sin))
plot: could not determine nonempty x axis; got: x-min = #f,
x-max = #f
```

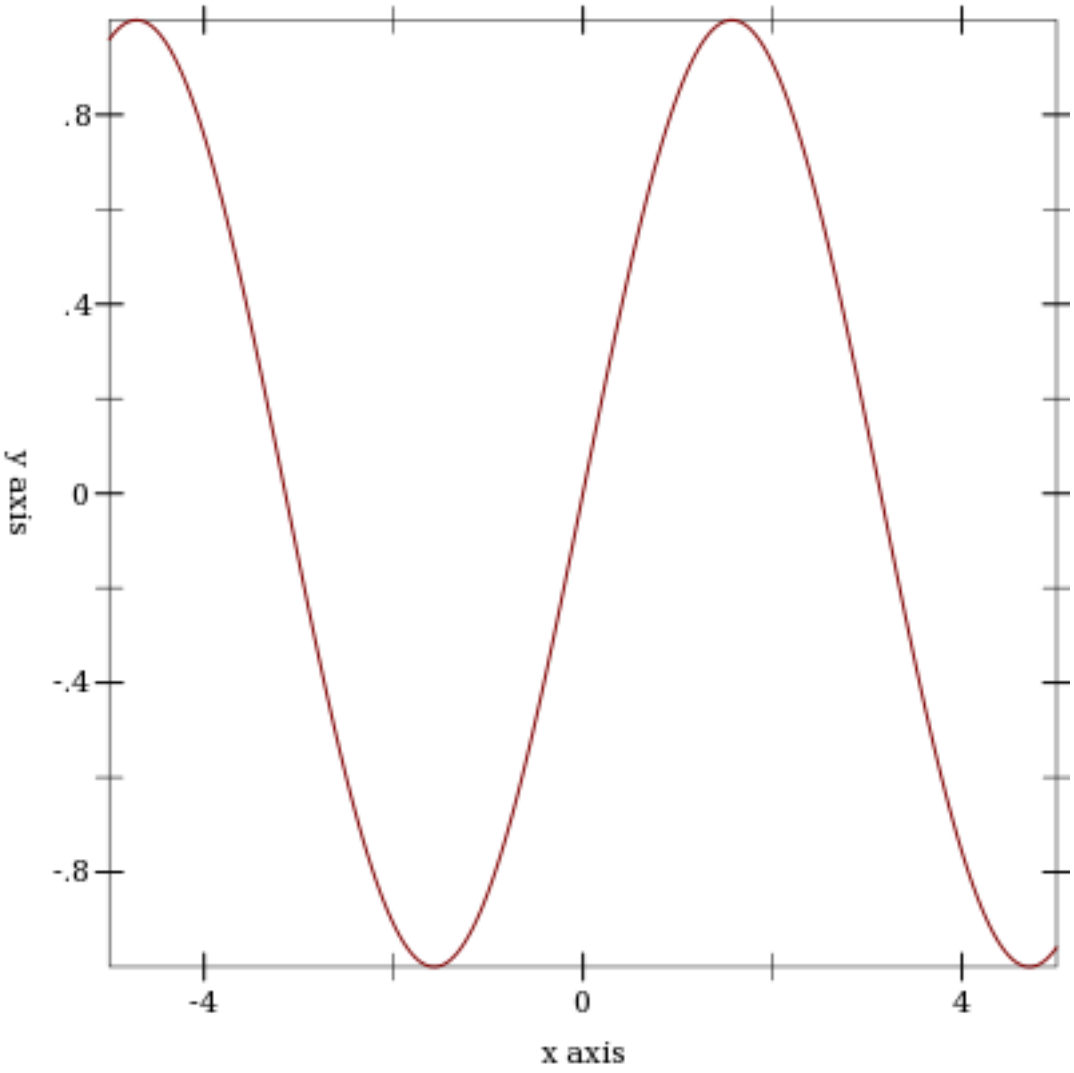
You could either change it to

```
> (plot (function sin) #:x-min -5 #:x-max 5 #:y-min -5 #:y-max 5)
```



or change it to

```
> (plot (function sin -5 5))
```



When `function` is given *x* bounds, it determines tight *y* bounds.

10.3 Changing Keyword Arguments

Replace every `#:width` in a call to `vector-field` with `#:line-width`.

Replace every `#:style 'scaled` with `#:scale 'auto` (or because it is the default in both the old and new, take it out).

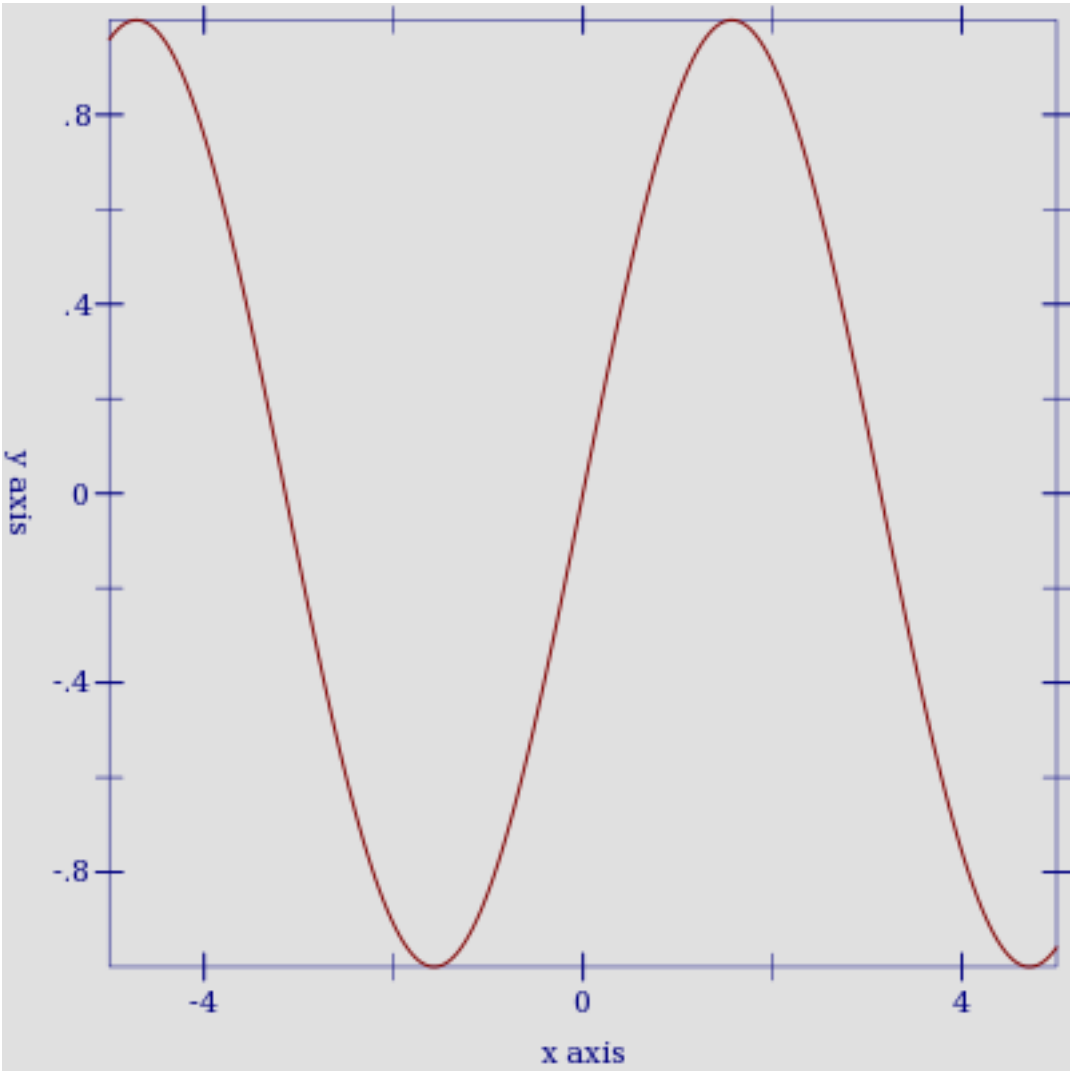
Replace every `#:style 'real` with `#:scale 1.0`.

Replace every `#:style 'normalized` with `#:scale 'normalized`.

The `plot` and `plot3d` functions still accept `#:bgcolor`, `#:fgcolor` and `#:lncolor`, but these are deprecated. Parameterize on `plot-background` and `plot-foreground` instead.

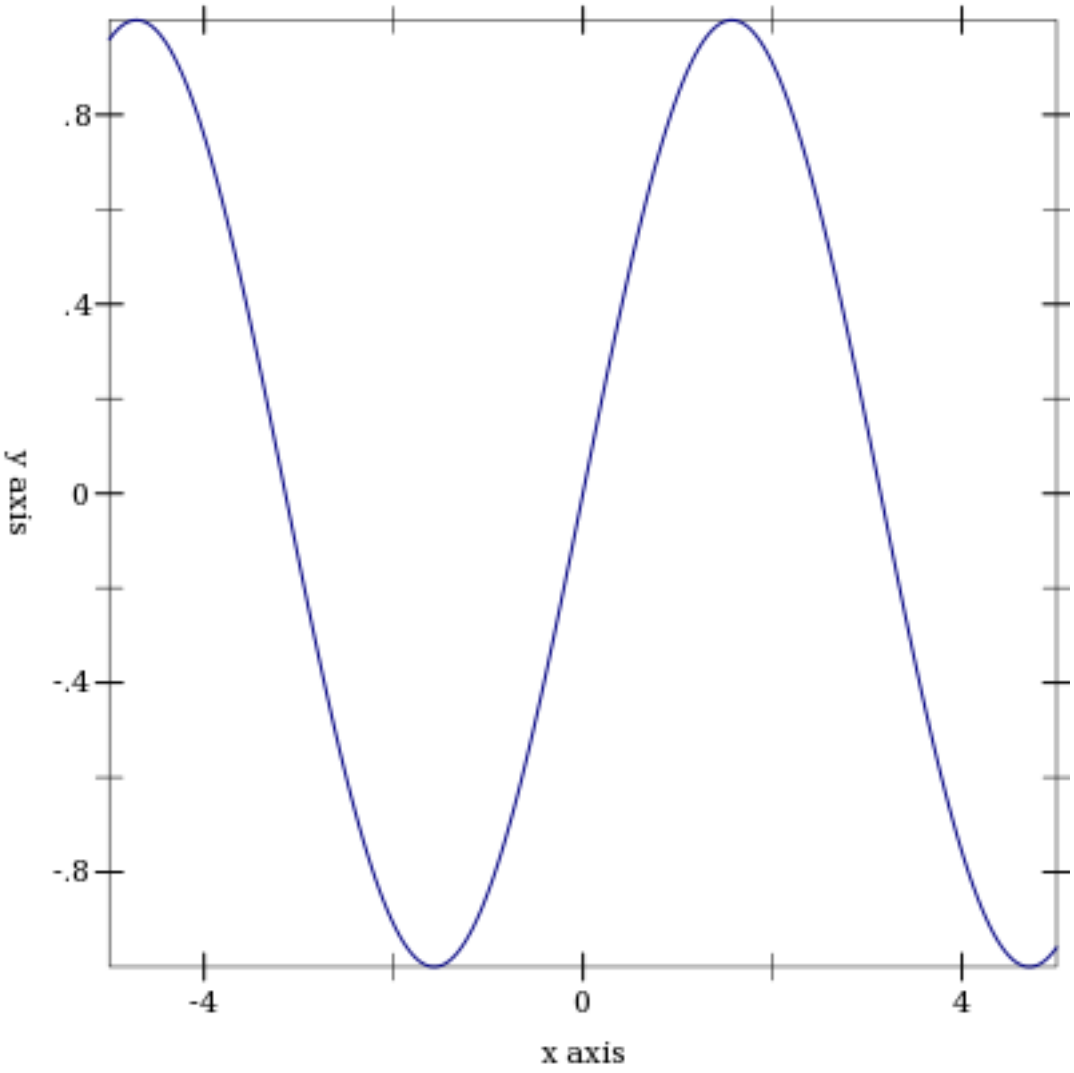
For example, if you have `(plot (function sin -5 5) #:fgcolor '(0 0 128) #:bgcolor '(224 224 224))`, change it to

```
> (parameterize ([plot-foreground  '(0 0 128)]
                 [plot-background  '(224 224 224)])
    (plot (function sin -5 5)))
```



The `#:lncolor` keyword argument now does nothing; change the renderer instead. For example, if you have `(plot (function sin -5 5) #:lncolor '(0 0 128))`, change it to

```
> (plot (function sin -5 5 #:color '(0 0 128)))
```



Change `#:az` in calls to `plot3d` to `#:angle`, and `#:alt` to `#:altitude`. Alternatively, parameterize multiple plots by setting the `plot3d-angle` and `plot3d-altitude` parameters.

10.4 Fixing Broken Calls to `points`

The `points` function used to be documented as accepting a `(listof (vector/c real? real?))`, but actually accepted a `(listof (vectorof real?))` and silently ignored any extra vector elements.

If you have code that takes advantage of this, strip down the vectors first. For example, if `vs` is the list of vectors, send `(map (λ (v) (vector-take v 2)) vs)` to `points`.

10.5 Replacing Uses of `plot-extend`

Chances are, if you used `plot-extend`, you no longer need it. The canonical `plot-extend` example used to be a version of `line` that drew dashed lines. Every line-drawing function in PLoT now has a `#:style` or `#:line-style` keyword argument.

The rewritten PLoT will eventually have a similar extension mechanism.

10.6 Deprecated Functions

The following functions exist for backward compatibility, but may be removed in the future. Set `(plot-deprecation-warnings? #t)` to be alerted the first time each is used.

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

See §11 “Compatibility Module” for the original documentation. Replace this with `list`.

```
(line  f
       [#:samples samples
        #:width width
        #:color color
        #:mode mode
        #:mapping mapping
        #:t-min t-min
        #:t-max t-max])    → renderer2d?
  f : (real? . -> . (or/c real? (vector/c real? real?)))
  samples : (and/c exact-integer? (>=/c 2)) = 150
  width : (>=/c 0) = 1
  color : plot-color/c = 'red
  mode : (one-of/c 'standard 'parametric) = 'standard
  mapping : (one-of/c 'cartesian 'polar) = 'cartesian
  t-min : real? = -5
  t-max : real? = 5
```

See §11 “Compatibility Module” for the original documentation. Replace this with `function`, `parametric` or `polar`, depending on keyword arguments.

```
(contour  f
          [#:samples samples
           #:width width
           #:color color
           #:levels levels]) → renderer2d?
  f : (real? real? . -> . real?)
  samples : (and/c exact-integer? (>=/c 2)) = 50
  width : (>=/c 0) = 1
  color : plot-color/c = 'black
  levels : (or/c (and/c exact-integer? (>=/c 2)) (listof real?))
         = 10
```

See §11 “Compatibility Module” for the original documentation. Replace this with contours.

```
(shade f [#:samples samples #:levels levels]) → renderer2d?
  f : (real? real? . -> . real?)
  samples : (and/c exact-integer? (>=/c 2)) = 50
  levels : (or/c (and/c exact-integer? (>=/c 2)) (listof real?))
         = 10
```

See §11 “Compatibility Module” for the original documentation. Replace this with contour-intervals.

```
(surface  f
          [#:samples samples
           #:width width
           #:color color])    → renderer3d?
  f : (real? real? . -> . real?)
  samples : (and/c exact-integer? (>=/c 2)) = 50
  width : (>=/c 0) = 1
  color : plot-color/c = 'black
```

See §11 “Compatibility Module” for the original documentation. Replace this with surface3d.

11 Compatibility Module

```
(require plot/compat)
```

This module provides an interface compatible with PLoT 5.1.3 and earlier.

Do not use both `plot` and `plot/compat` in the same module. It is tempting to try it, to get both the new features and comprehensive backward compatibility. But to enable the new features, the objects plotted in `plot` have to be a different data type than the objects plotted in `plot/compat`. They do not coexist easily, and trying to make them do so will result in contract violations.

11.1 Plotting

```
(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 image-snip%)
  data : ((is-a?/c 2d-plot-area%) . -> . 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 2D, where *data* is generated by functions like `points` or `line`.

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

The result is a image-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.

The #:lncolor keyword argument is accepted for backward compatibility, but does nothing.

```
(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
          #:out-file out-file]) → (is-a?/c image-snip%)
  data : ((is-a?/c 3d-plot-area%) . -> . 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)
  out-file : (or/c path-string? output-port? #f) = #f
```

Plots *data* in 3D, 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 3D *data* value is represented as a procedure that takes a 3d-plot-area% instance and adds plot information to it.

The #:lncolor keyword argument is accepted for backward compatibility, but does nothing.

```
(points vecs [#:sym sym #:color color])
 → ((is-a?/c 2d-plot-area%) . -> . void?)
  vecs : (listof (vectorof real?))
  sym : (or/c char? string? exact-integer? symbol?) = 'square
  color : plot-color? = 'black
```

Creates 2D 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, 'leftarrow, 'rightarrow, 'uparrow, 'downarrow.

```
(line f
      [#:samples samples
       #:width width
       #:color color
       #:mode mode
       #:mapping mapping
       #:t-min t-min
       #:t-max t-max])
 → ((is-a?/c 2d-plot-area%) . -> . void?)
  f : (real? . -> . (or/c real? (vector/c real? real?)))
  samples : (and/c exact-integer? (>=/c 2)) = 150
  width : (>=/c 0) = 1
  color : plot-color/c = 'red
  mode : (one-of/c 'standard 'parametric) = 'standard
  mapping : (one-of/c 'cartesian 'polar) = 'cartesian
  t-min : real? = -5
  t-max : real? = 5
```

Creates 2D 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-plot-area%) . -> . void?)
  vecs : (listof (vector/c real? real? real?))
  color : plot-color? = 'black
```

Creates 2D 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
               [#:samples samples
                #:width width
                #:color color
                #:style style])
 → ((is-a?/c 2d-plot-area%) . -> . void?)
  f : ((vector/c real? real?) . -> . (vector/c real? real?))
  samples : (and/c exact-integer? (>=/c 2)) = 20
  width : exact-positive-integer? = 1
  color : plot-color? = 'red
  style : (one-of/c 'scaled 'normalized 'real) = 'scaled
```

Creates 2D 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-plot-area%) . -> . void?)
  f : (real? real? . -> . real?)
  samples : exact-nonnegative-integer? = 50
  width : (>=/c 0) = 1
  color : plot-color/c = 'black
  levels : (or/c (and/c exact-integer? (>=/c 2)) (listof real?))
         = 10
```

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

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

Creates 2D 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-plot-area%) . -> . void?)
  f : (real? real? . -> . real?)
  samples : (and/c exact-integer? (>=/c 2)) = 50
  width : (>=/c 0) = 1
  color : plot-color/c = 'black
```

Creates 3D plot data to draw a 3D surface in a 2D 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
```

11.2 Curve Fitting

Do not use the fit function. It is going to be removed in Racket 5.2.1. It relies on old C code that nobody understands or is willing to maintain, and that is also slightly crashy.

Quite independent of plotting, and for reasons lost in the sands of time, the plot library provides a non-linear, least-squares fit algorithm to fit parameterized functions to given data. The code that implements the algorithm is public domain, and is used by the gnuplot package.

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)))
```

- 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.

```
(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 `"compat/tests/fit-demo-2.rkt"` in the `"plot"` collection.

```
(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?)))
```

Do not use the `fit` function. It is going to be removed in Racket 5.2.1. It relies on old C code that nobody understands or is willing to maintain, and that is also slightly crashy.

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.

```
(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.

11.3 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.

12 To Do

- Planned new renderers
  - 2D kernel density estimator
  - 3D kernel density estimator
  - 2D implicit curve
  - 3D implicit surface
  - 3D decorations: labeled points, axes, grids
- Possible new renderers
  - R × R -> R parametric (turn into 3D implicit surface by solving for minimum distance?)
  - 3D vector field
  - Head-to-tail vector fields
- Minor fixes
  - Subdivide nonlinearly transformed 3D lines/polygons (port from `2d-plot-area%`)
- Minor enhancements
  - Better depth sorting (possibly split intersecting polygons; look into BSP tree)
  - Legend entries have minimum sizes
  - Log-scale tick functions (i.e. major ticks are 10^0, 10^1, 10^2, ...)
  - Label contour heights on the contour lines
  - 3D support for exact rational functions (i.e. polynomial at [big..big+ϵ])
  - Join 2D contour lines
  - More appearance options (i.e. draw 2D tick labels on right/top)
  - Manually exclude discontinuous points from function renderers: allow values `(hole p1 p2)`, `(left-hole p1 p2)`, `(right-hole p1 p2)`
  - `histogram-list` to plot multiple histograms without manually calculating `#:x-min`