PLoT: Graph Plotting

Version 5.3

Neil Toronto <neil.toronto@gmail.com>

August 6, 2012

```
(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 §11 "Porting From PLoT <= 5.1.3" (and possibly §12 "Compatibility Module").

Contents

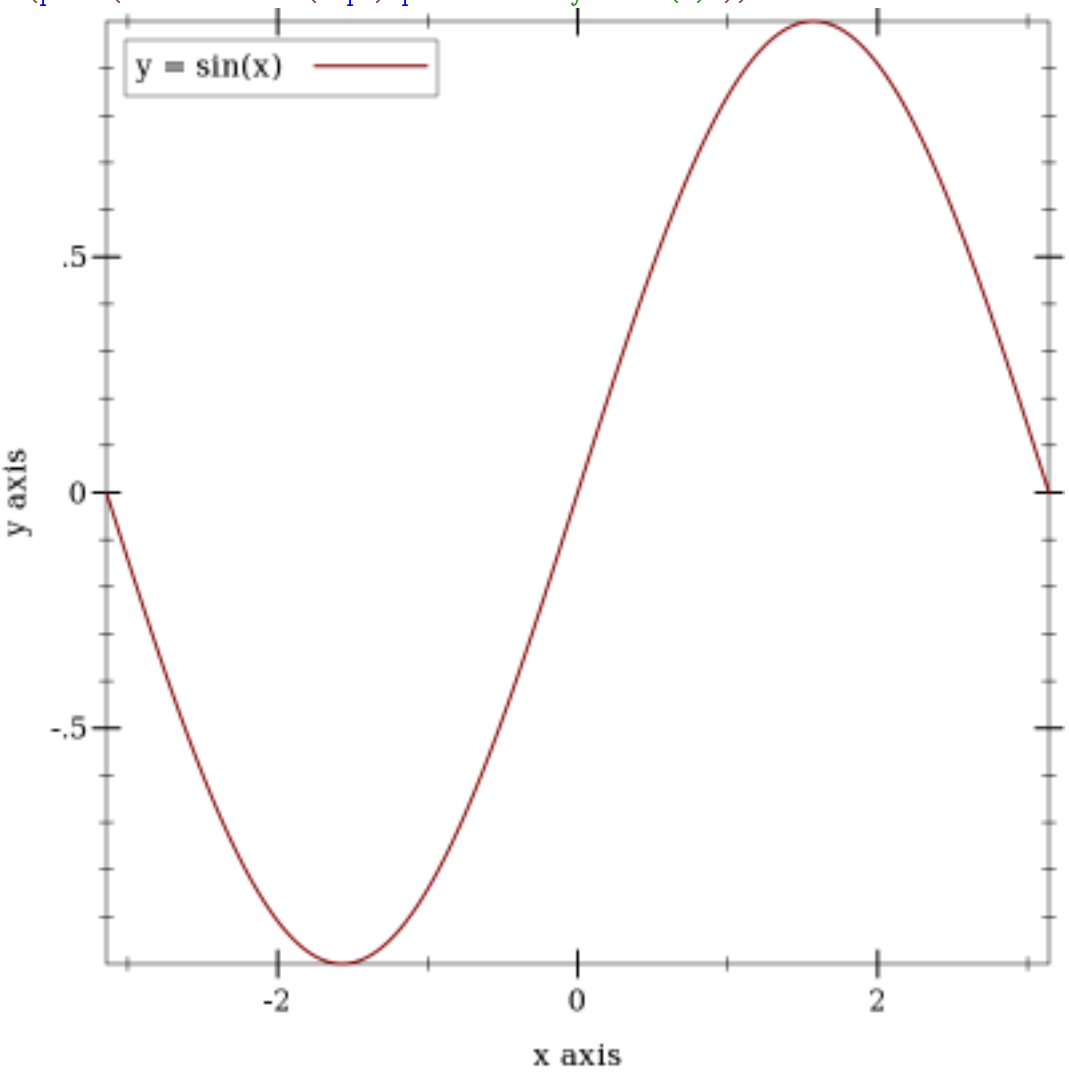
1 Introduction

1.1 Plotting 2D Graphs

To plot a one-input, real-valued function, do something like

```
> (require plot)
```

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



(If you're not using DrRacket, start with

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

to open each plot in a new window.)

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.

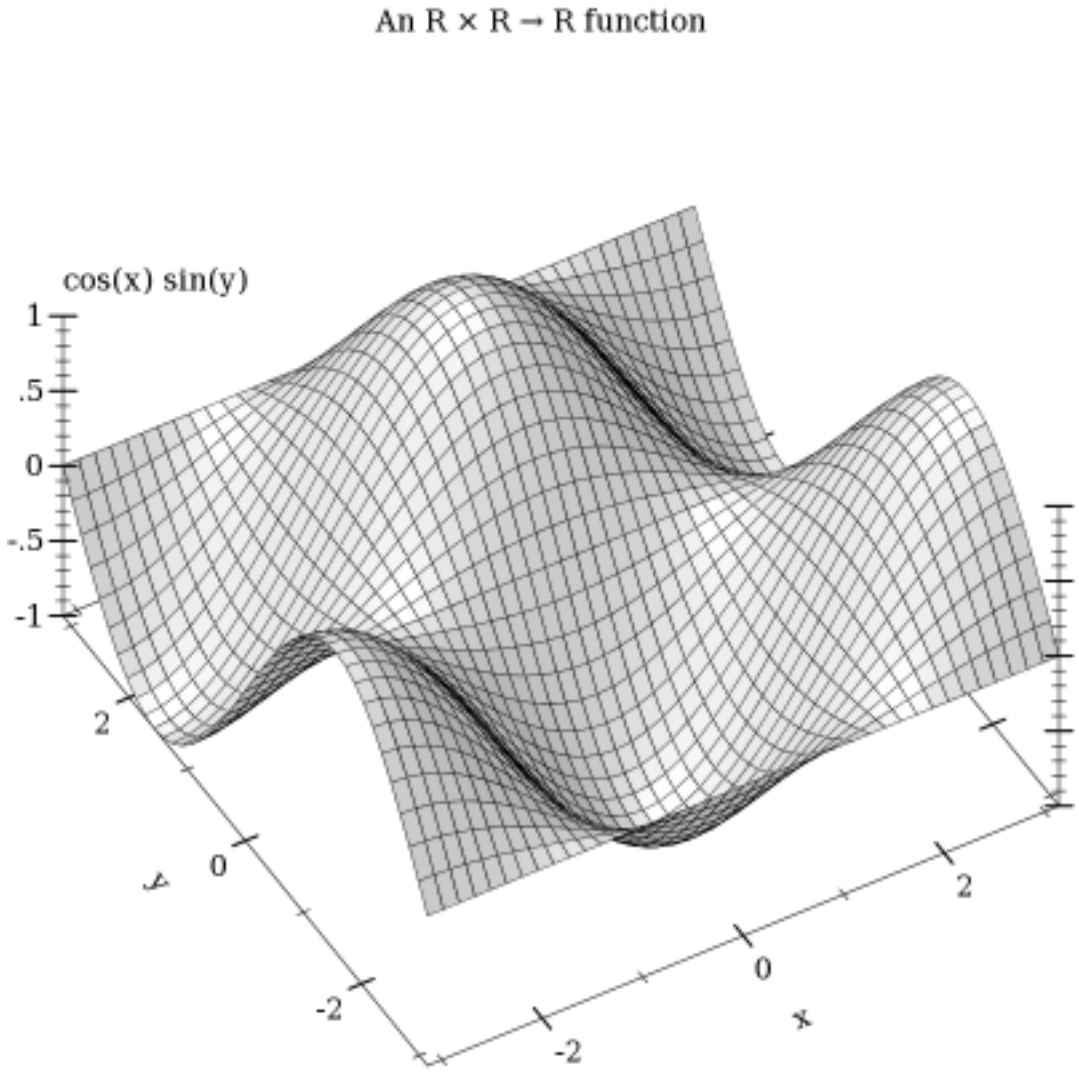
The `plot` function collects legend entries, bounds and ticks. It then sets up a *plot area* with large enough bounds to contain the renderers, draws the axes and ticks, invokes the renderers' drawing procedures, and then draws the legend.

1.3 Plotting 3D Graphs

To plot a two-input, real-valued function as a surface, 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!

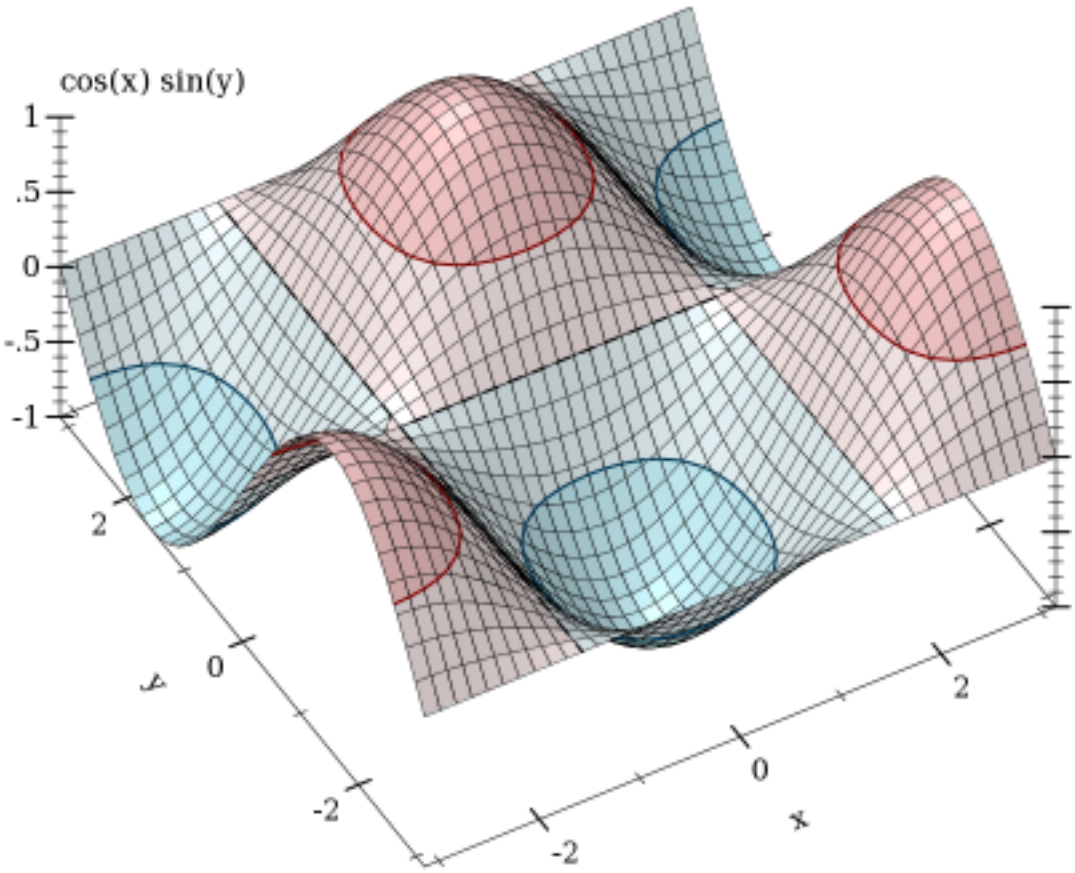


This example also demonstrates using keyword arguments that change the plot, such as `#:title`. In PLoT, every keyword argument is optional and almost all have parameterized default values. In the case of `plot3d`'s `#:title`, the corresponding parameter is `plot-title`. That is, keyword arguments are 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 §9 "Plot and Renderer Parameters" for the full listing.

```
                (- pi) pi (- pi) pi)))
```
An R × R → R function



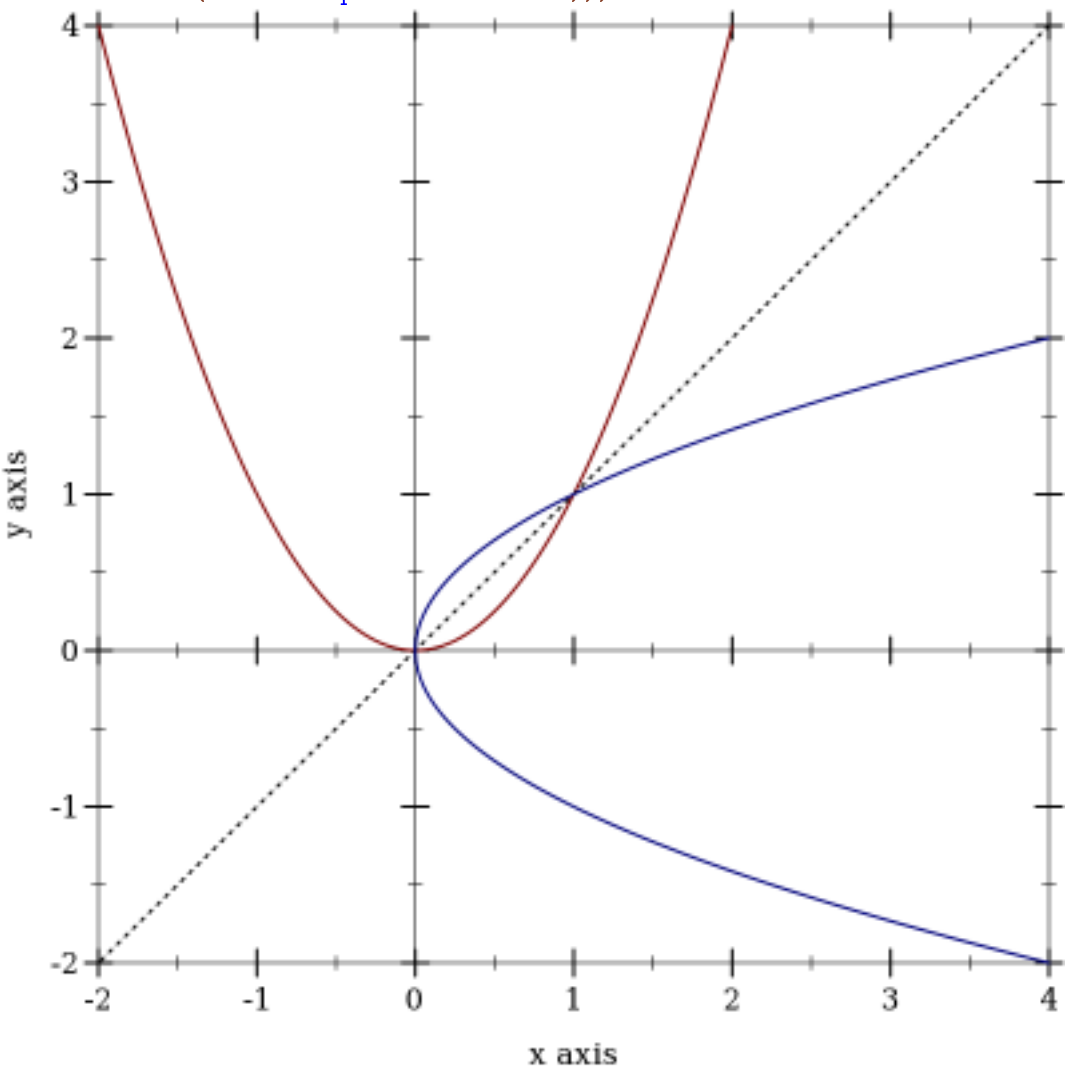
This example also demonstrates `contour-intervals3d`, which colors the surface between contour lines, or lines of constant height. By default, `contour-intervals3d` places the contour lines at the same heights as the ticks on the z axis.

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)))
```



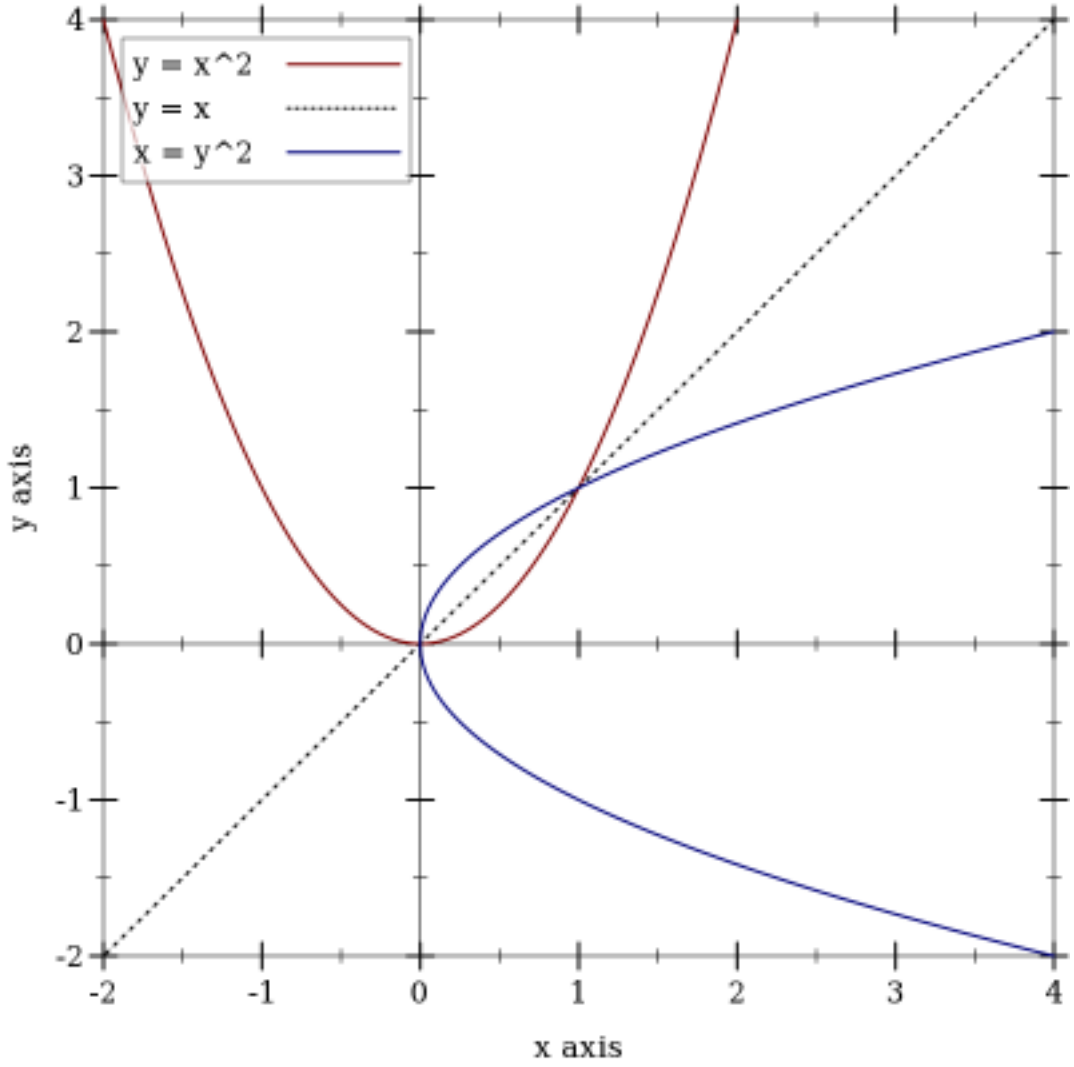
Here, `inverse` plots the inverse of a function. (Both `function` and `inverse` plot the reflection line `(λ (x) x)` identically.)

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.

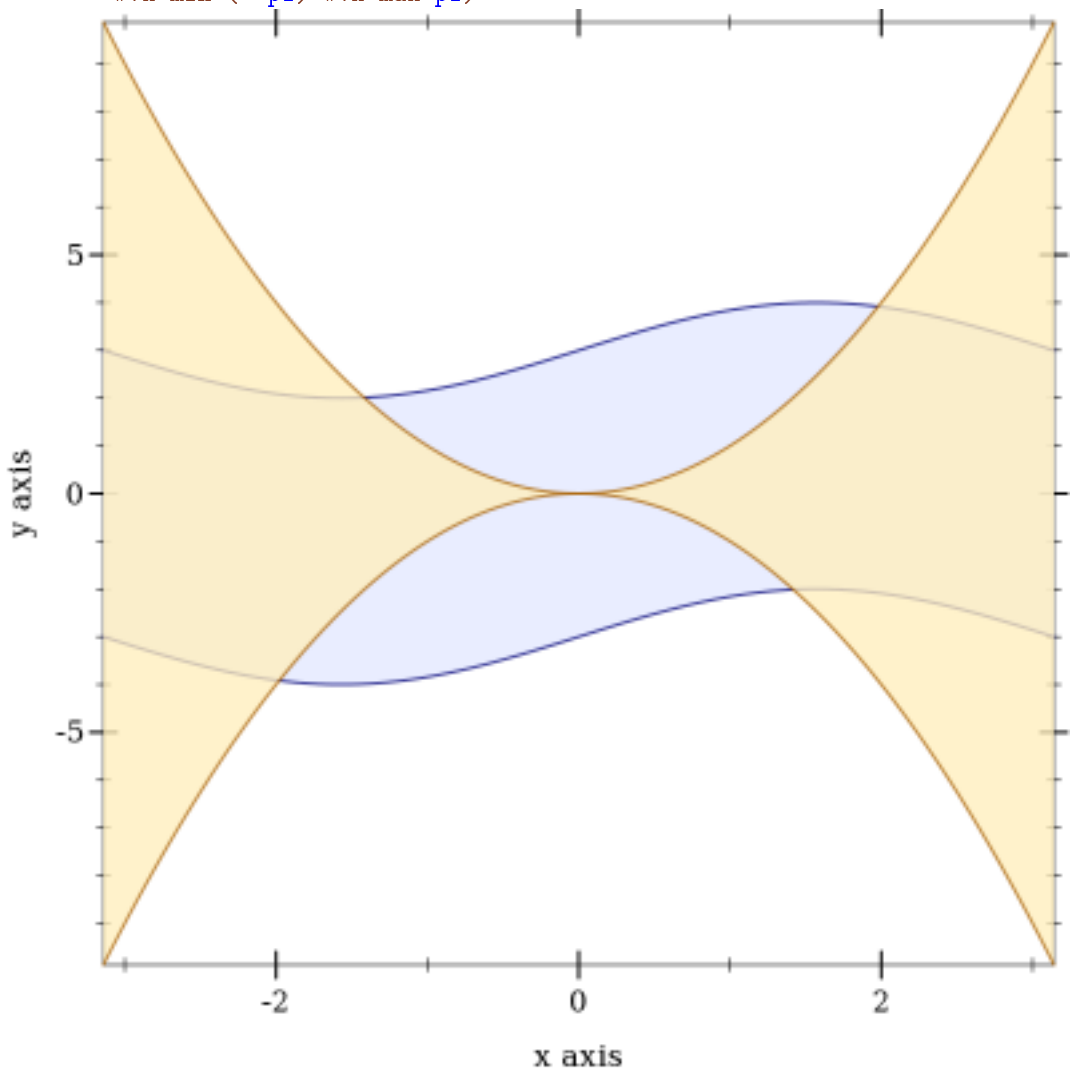
Renderers generate legend entries when passed a `#:label` argument. For example,

```
> (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.

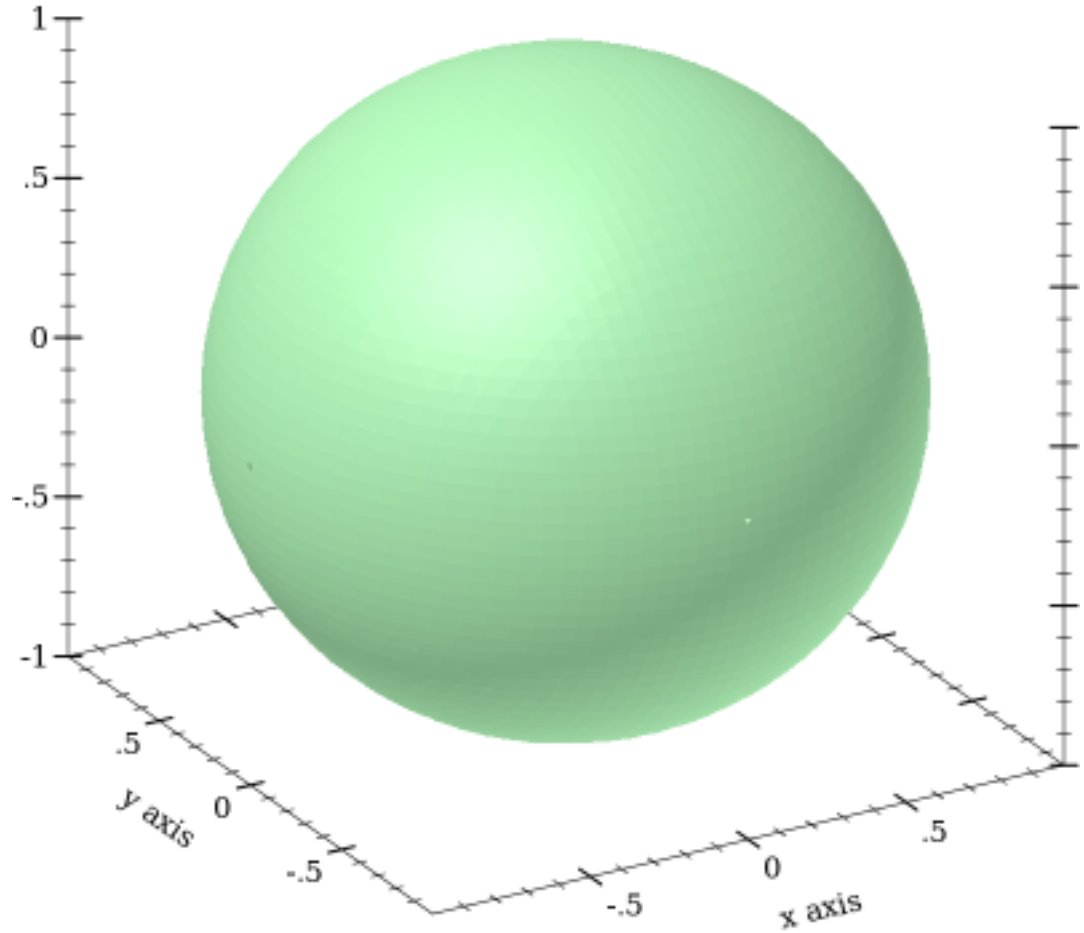
It is not always possible for renderers and `plot` or `plot3d` to determine the bounds:

```
> (plot (function sqr))
plot: could not determine sensible plot bounds; got x ∈
[#f,#f], y ∈ [#f,#f]
> (plot (function sqr #f #f))
plot: could not determine sensible plot bounds; got x ∈
[#f,#f], y ∈ [#f,#f]
> (plot (function sqr (- pi)))
plot: could not determine sensible plot bounds; got x ∈
[-3.141592653589793,#f], y ∈ [#f,#f]
> (plot (list (function sqr #f 0)
              (function sqr 0 #f)))
plot: could not determine sensible plot bounds; got x ∈
[0,0], y ∈ [0,0]
```

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*.

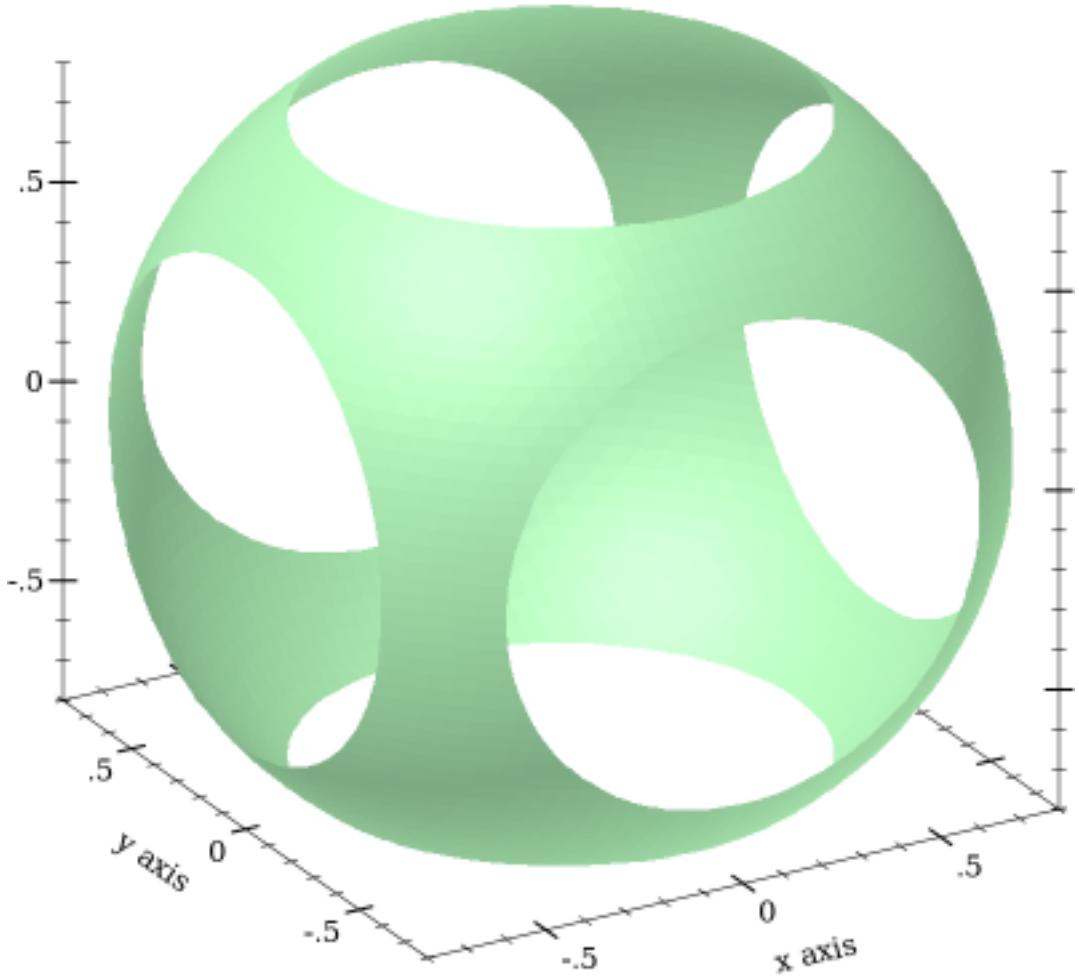
Here is an example of commanding `plot3d` to override a renderer's bounds. First, consider the plot of a sphere with radius `1`:

```
> (plot3d (polar3d (λ (θ ρ) 1) #:color 2 #:line-
style 'transparent)
          #:altitude 25)
```



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

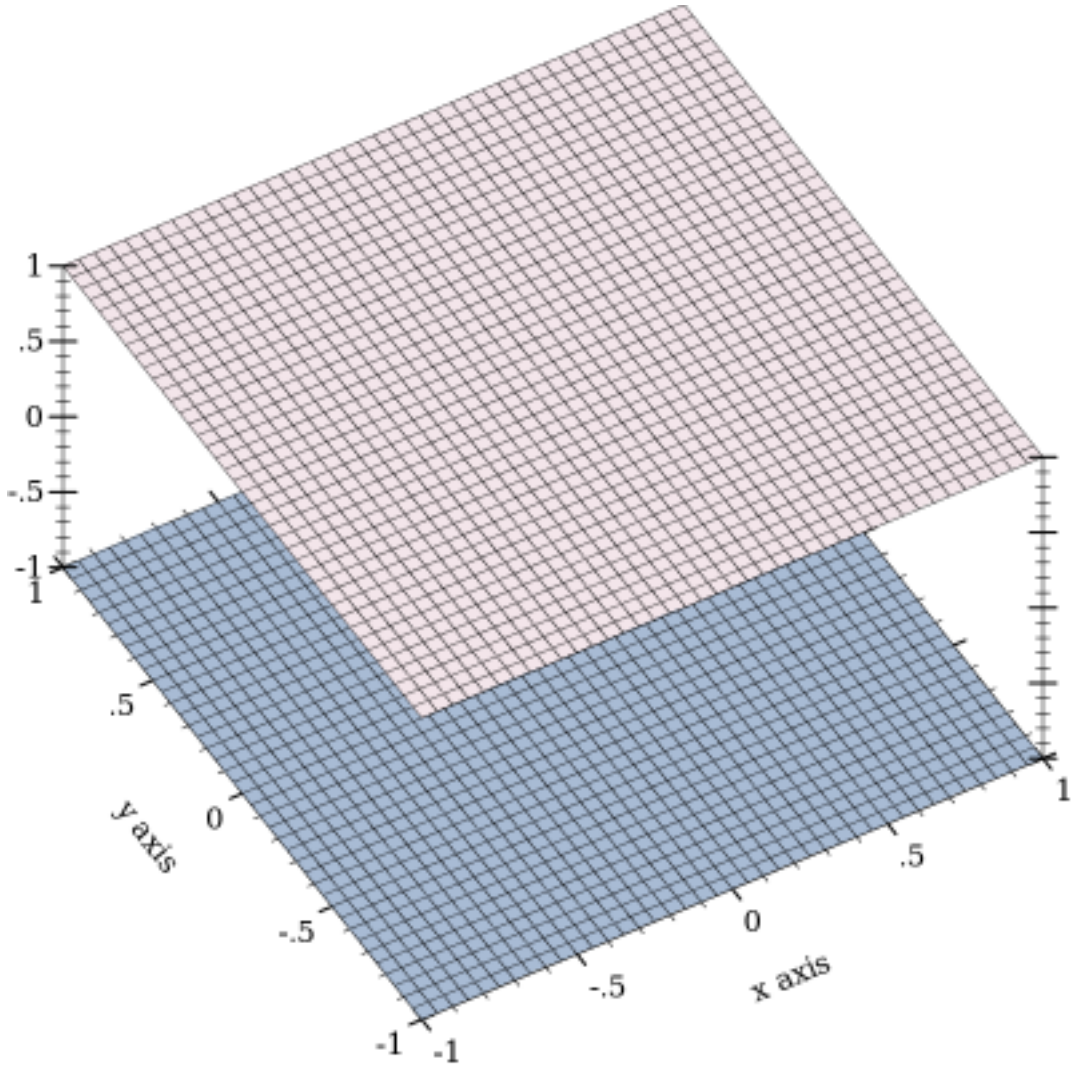
```
> (plot3d (polar3d (λ (θ ρ) 1) #:color 2 #:line-
style 'transparent)
          #:x-min -0.8 #:x-max 0.8
          #:y-min -0.8 #:y-max 0.8
          #:z-min -0.8 #:z-max 0.8
          #:altitude 25)
```



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

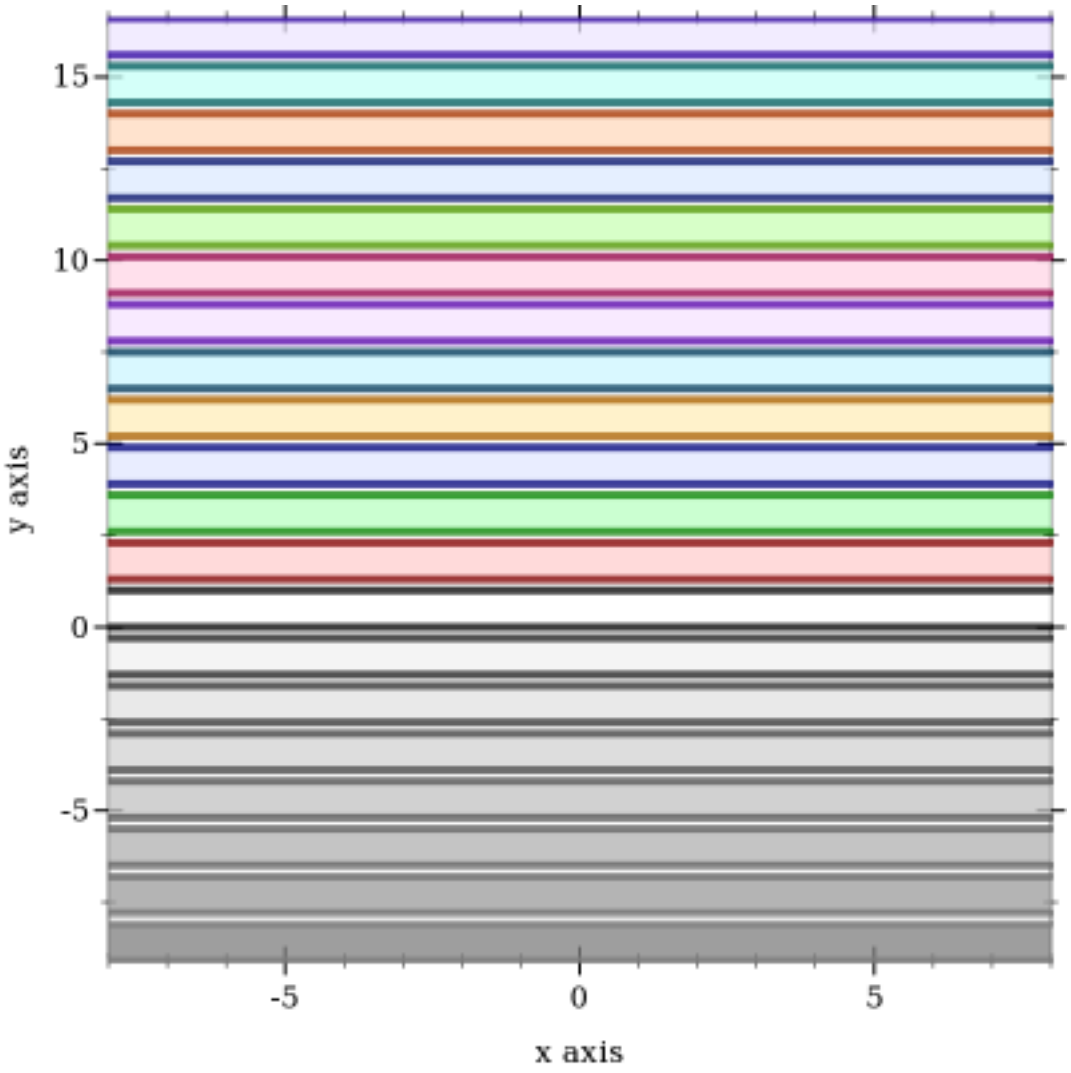
Any plot can be rendered to PNG, PDF, PS and SVG files using `plot-file` and `plot3d-file`, to include in papers and other published media.

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 -7 13)])
            (function-interval
             (λ (x) (* i 1.3)) (λ (x) (+ 1 (* i 1.3)))
             #:color i #:line1-color i #:line2-color i))
          #:x-min -8 #:x-max 8))
```



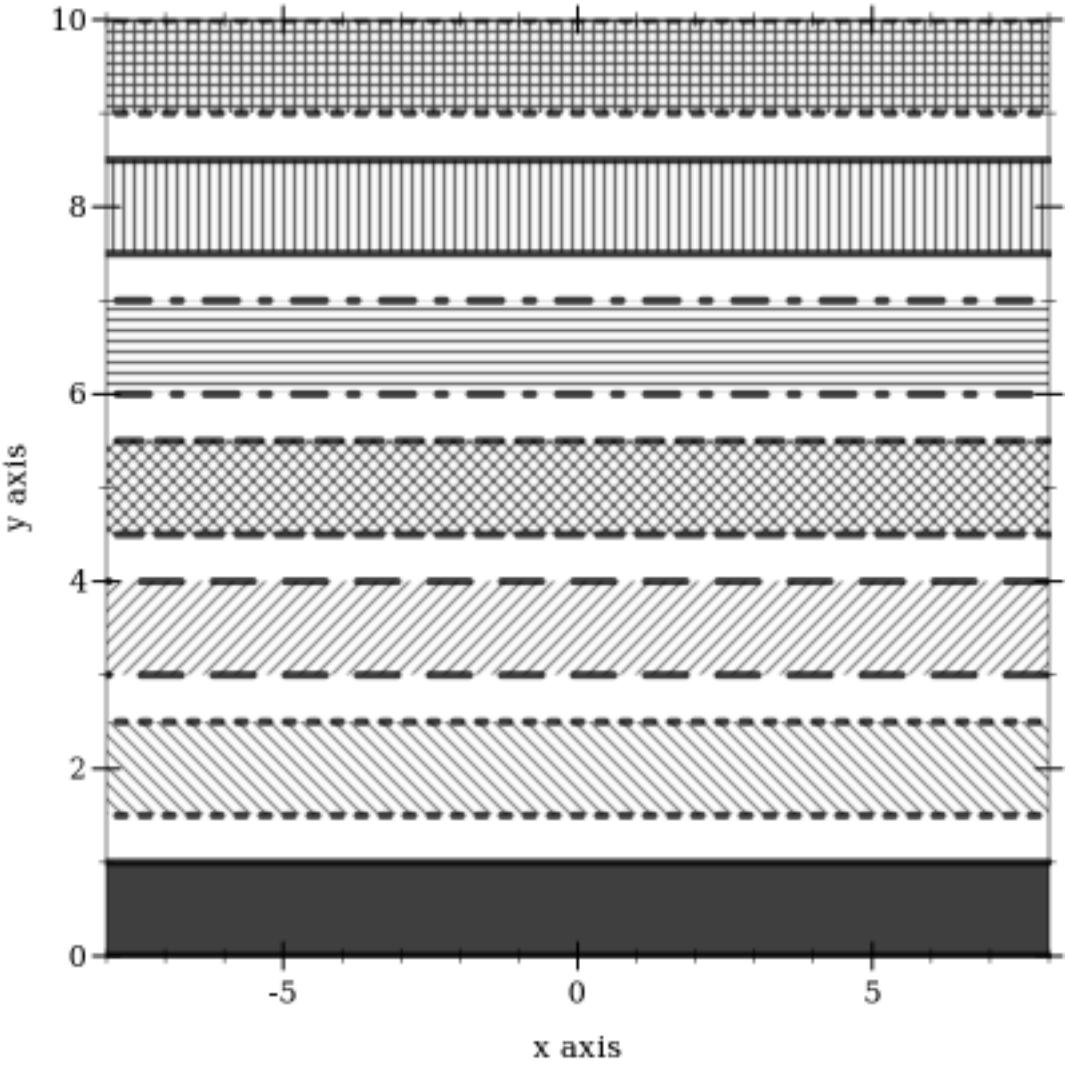
Color `0` is black for lines and white for areas. Colors `1`..`120` are generated by rotating hues and adjusting to make neighbors more visually dissimilar. Colors `121`..`127` are grayscale.

Colors `-7`..`-1` are also grayscale because before `0`, colors repeat. That is, colors `-128`..`-1` are identical to colors `0`..`127`. Colors also repeat after `127`.

If the paper will be published in black and white, use styles as well as, or instead of, colors. 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))
```



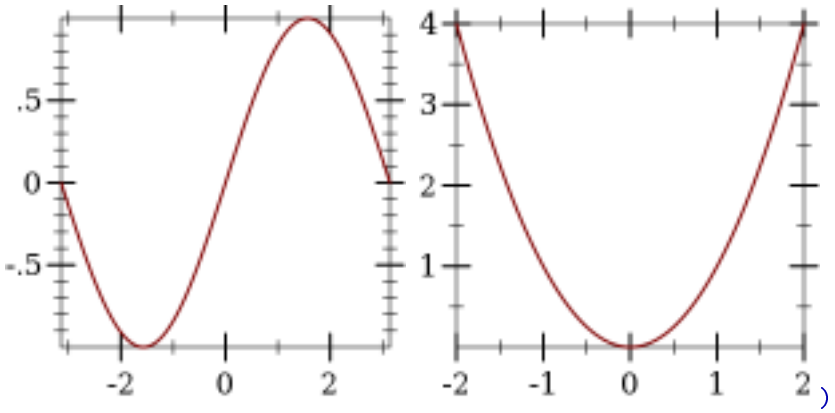
2 2D Plot Procedures

```
(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 (or/c renderer2d? nonrenderer?))
  x-min : (or/c rational? #f) = #f
  x-max : (or/c rational? #f) = #f
  y-min : (or/c rational? #f) = #f
  y-max : (or/c rational? #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 

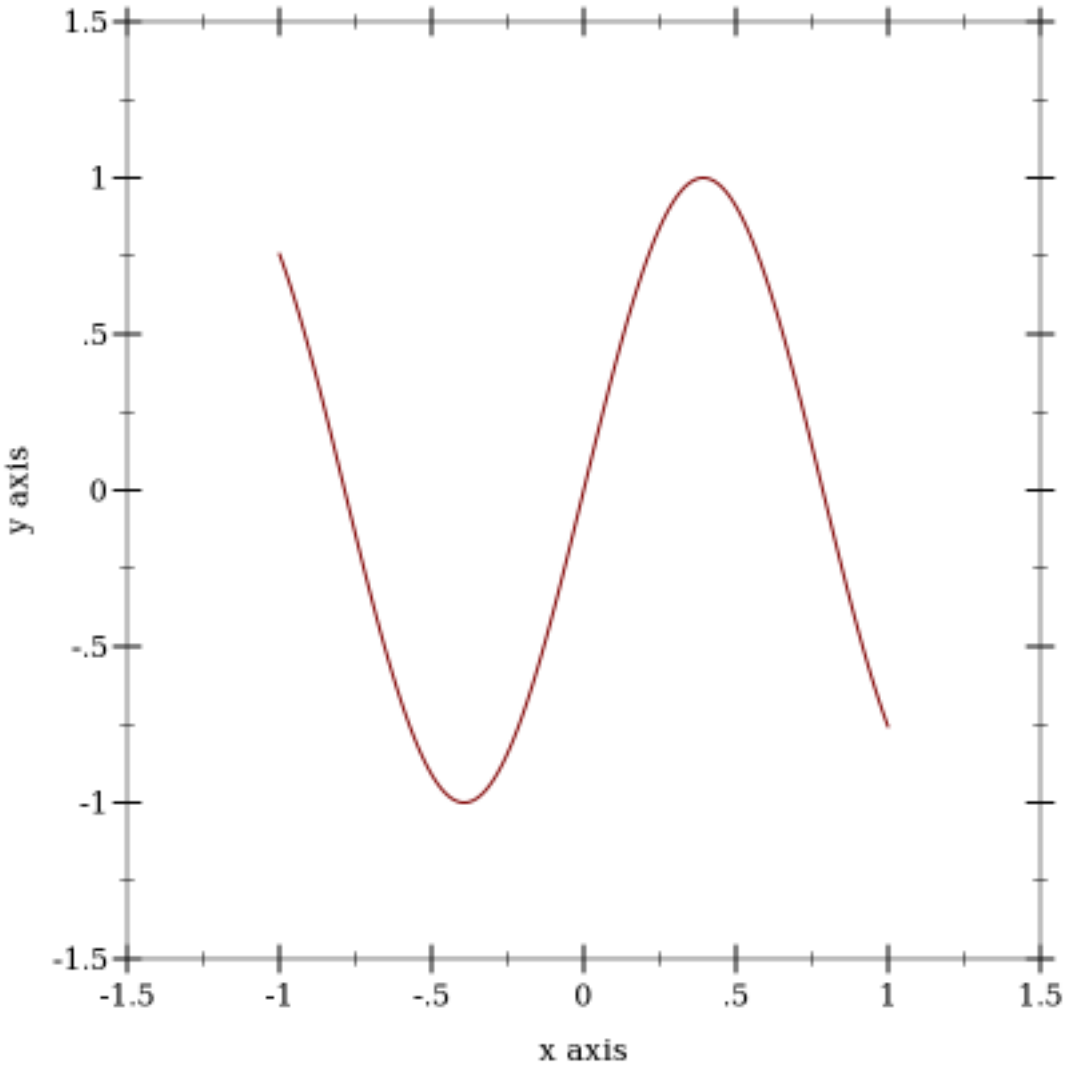
)

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 (or/c renderer2d? nonrenderer?))
  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 (or/c renderer2d? nonrenderer?))
(plot-bitmap renderer-tree ...) → (is-a?/c bitmap%)
  renderer-tree : (treeof (or/c renderer2d? nonrenderer?))
(plot-snip renderer-tree ...) → (is-a?/c image-snip%)
  renderer-tree : (treeof (or/c renderer2d? nonrenderer?))
(plot-frame renderer-tree ...) → (is-a?/c frame%)
  renderer-tree : (treeof (or/c renderer2d? nonrenderer?))
```

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 interactive 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 (or/c renderer2d? nonrenderer?))
  dc : (is-a?/c dc<%>)
  x : real?
  y : real?
  width : (>=/c 0)
  height : (>=/c 0)
  x-min : (or/c rational? #f) = #f
  x-max : (or/c rational? #f) = #f
  y-min : (or/c rational? #f) = #f
  y-max : (or/c rational? #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

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

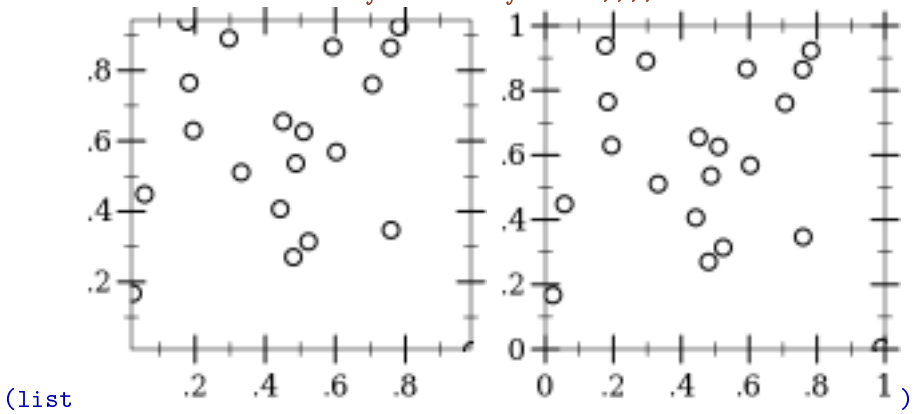
```
(points  vs
         [#:x-min x-min
          #:x-max x-max
          #:y-min y-min
          #:y-max y-max
          #:sym sym
          #:color color
          #:fill-color fill-color
          #:size size
          #:line-width line-width
          #:alpha alpha
          #:label label])          → renderer2d?
  vs : (listof (vector/c real? real?))
  x-min : (or/c rational? #f) = #f
  x-max : (or/c rational? #f) = #f
  y-min : (or/c rational? #f) = #f
  y-max : (or/c rational? #f) = #f
  sym : point-sym/c = (point-sym)
  color : plot-color/c = (point-color)
  fill-color : (or/c plot-color/c 'auto) = 'auto
  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 rational? #f) = #f
  x-max : (or/c rational? #f) = #f
  y-min : (or/c rational? #f) = #f
  y-max : (or/c rational? #f) = #f
  samples : exact-positive-integer? = (vector-field-samples)
```

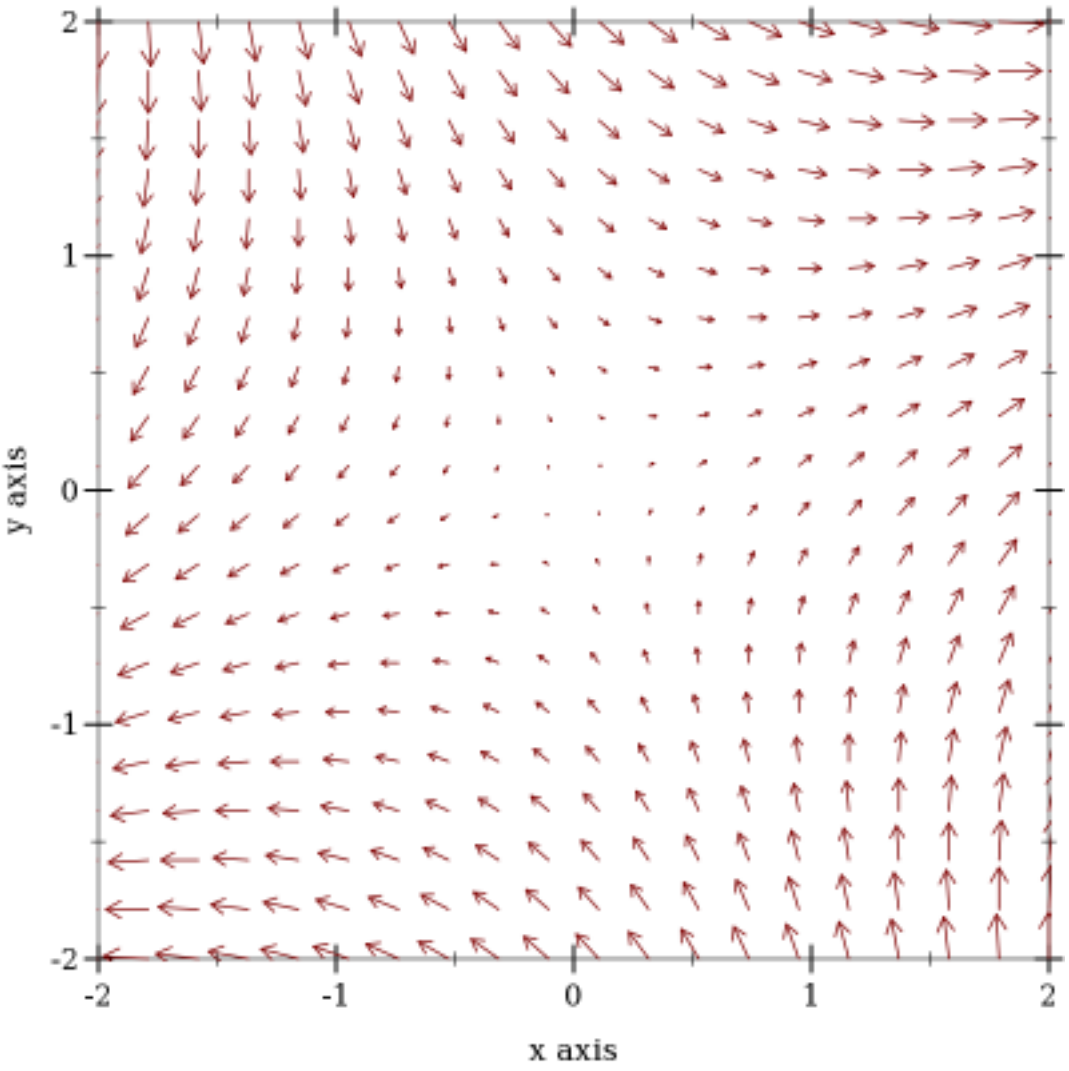
```
scale : (or/c real? (one-of/c 'auto 'normalized))
      = (vector-field-scale)
color : plot-color/c = (vector-field-color)
line-width : (>=/c 0) = (vector-field-line-width)
line-style : plot-pen-style/c = (vector-field-line-style)
alpha : (real-in 0 1) = (vector-field-alpha)
label : (or/c string? #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))
```

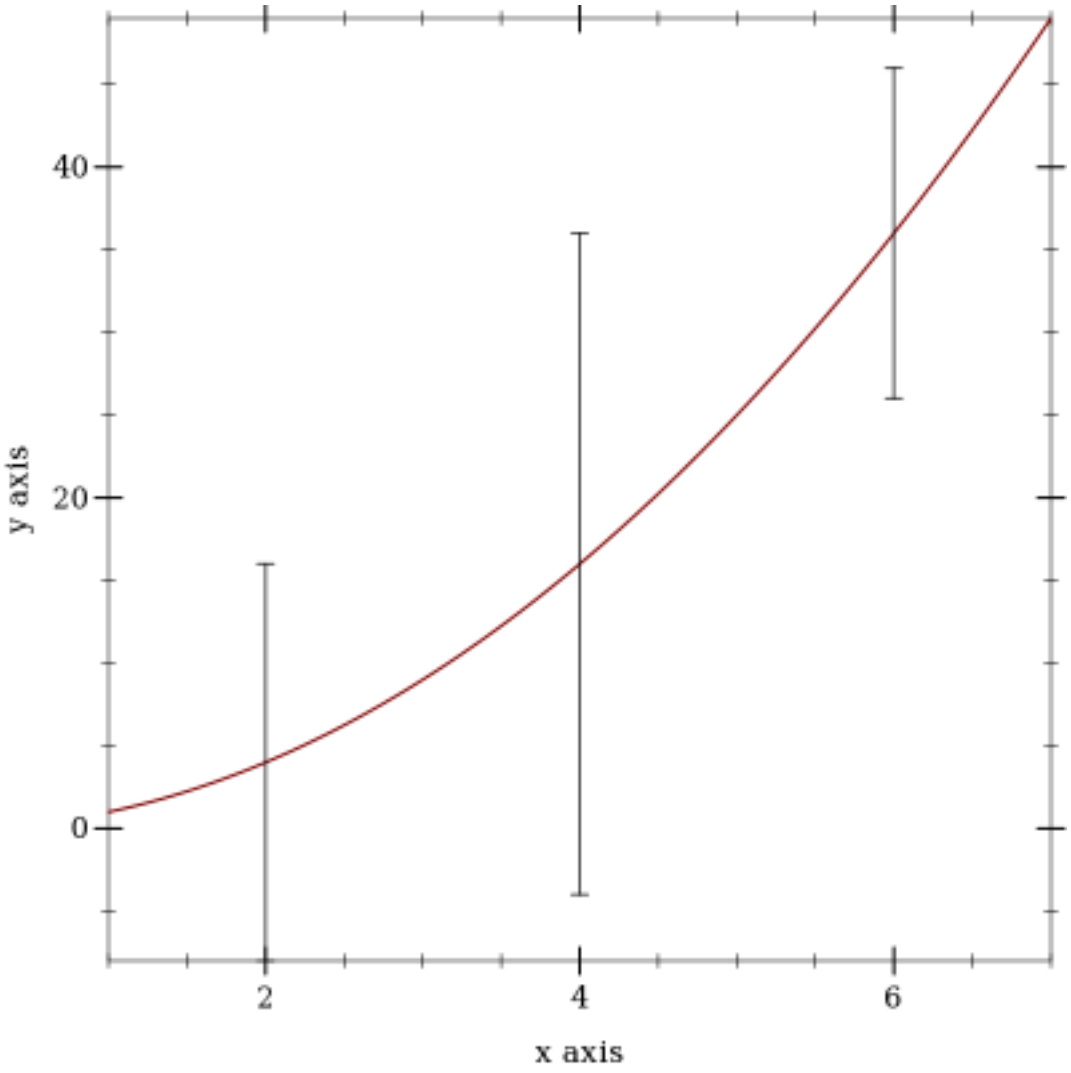



```
(error-bars  bars
             [#:x-min x-min
              #:x-max x-max
              #:y-min y-min
              #:y-max y-max
              #:color color
              #:line-width line-width
              #:line-style line-style
              #:width width
              #:alpha alpha])     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

```
x-min : (or/c rational? #f) = #f
x-max : (or/c rational? #f) = #f
y-min : (or/c rational? #f) = #f
y-max : (or/c rational? #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)))))
```

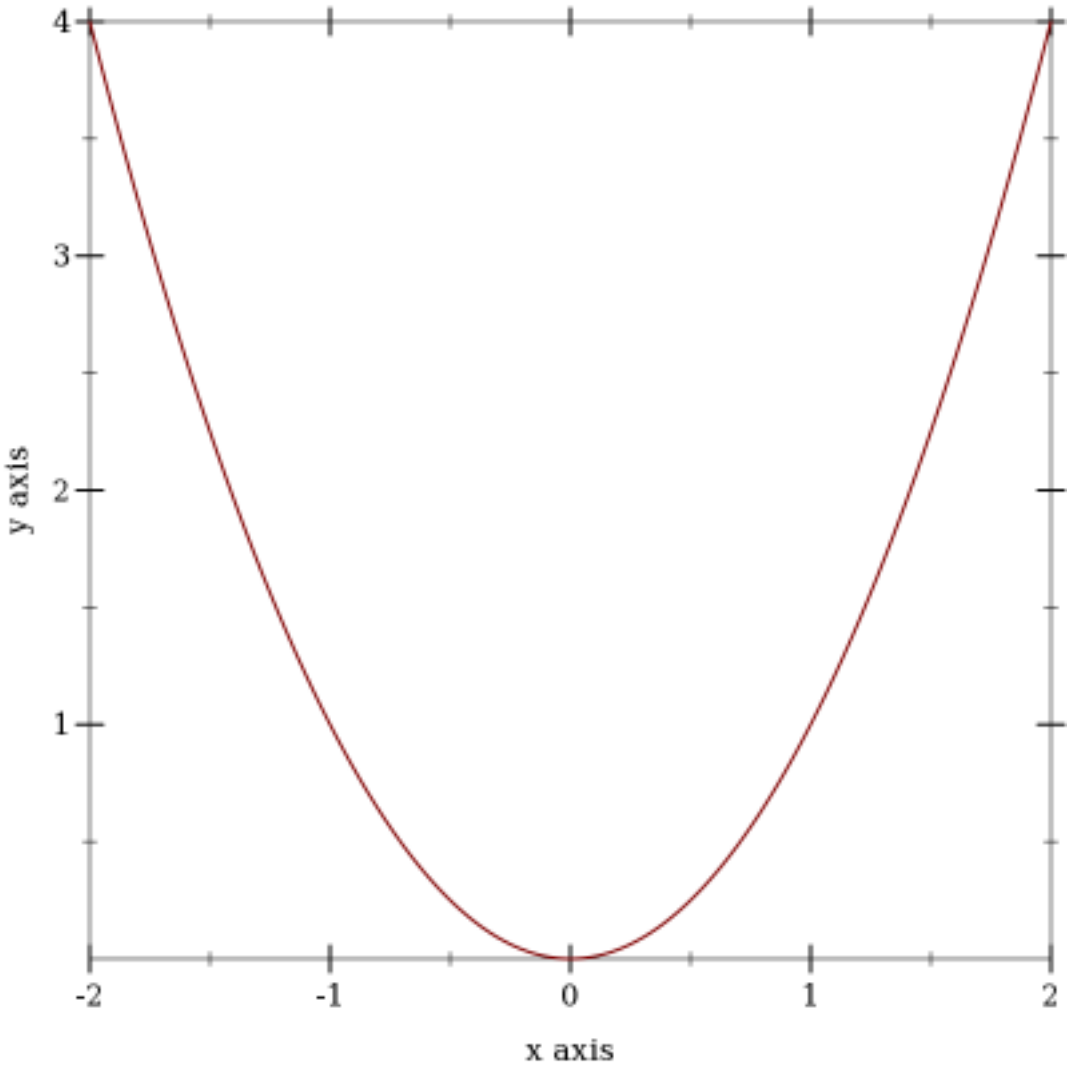


3.3 2D Line Renderers

```
(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 rational? #f) = #f
  x-max : (or/c rational? #f) = #f
  y-min : (or/c rational? #f) = #f
  y-max : (or/c rational? #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))
```

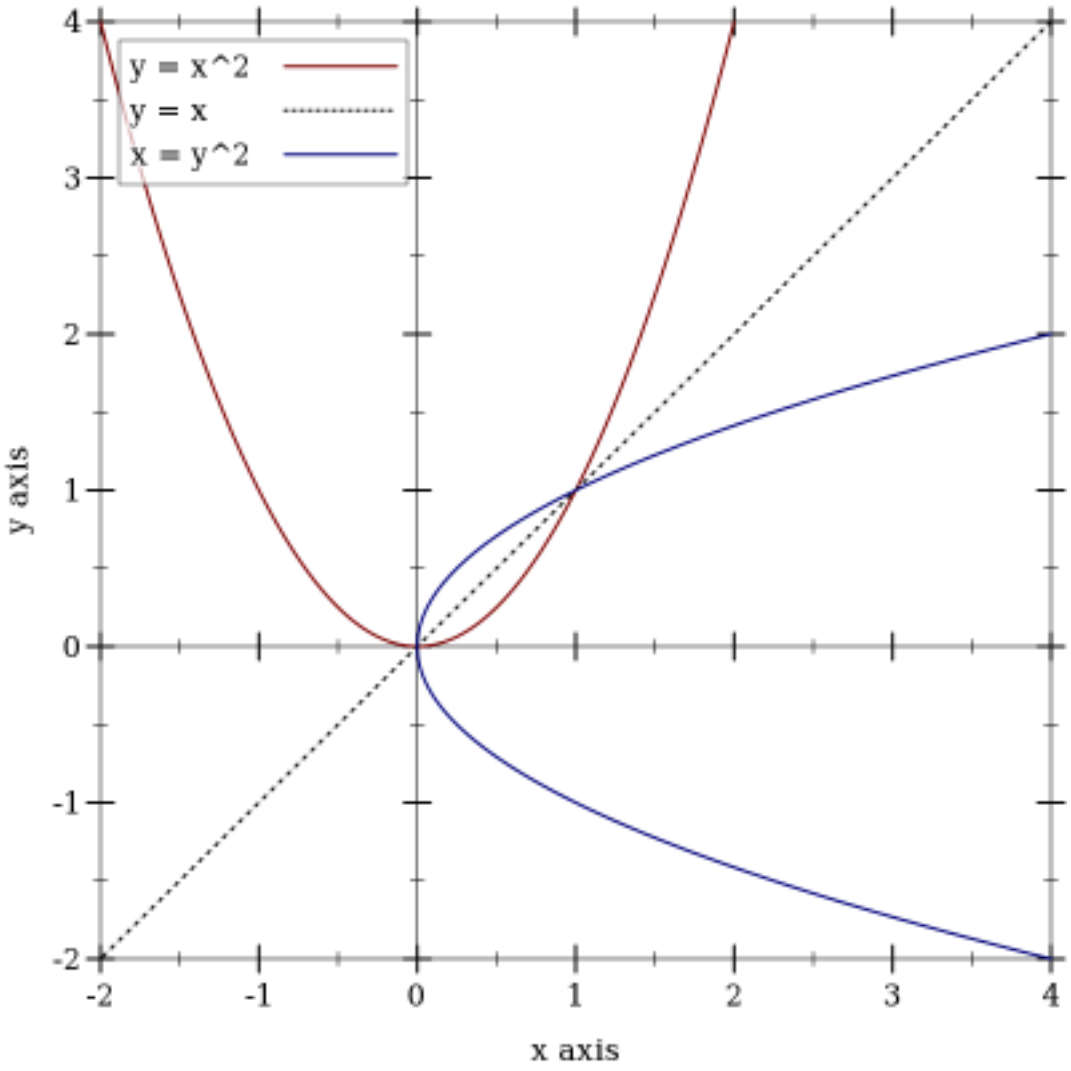



```
(inverse  f
          [y-min
           y-max
           #:x-min x-min
           #:x-max x-max
           #:samples samples
           #:color color
           #:width width
           #:style style
           #:alpha alpha
           #:label label])    → renderer2d?
```

```
f : (real? . -> . real?)
y-min : (or/c rational? #f) = #f
y-max : (or/c rational? #f) = #f
x-min : (or/c rational? #f) = #f
x-max : (or/c rational? #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")))
```

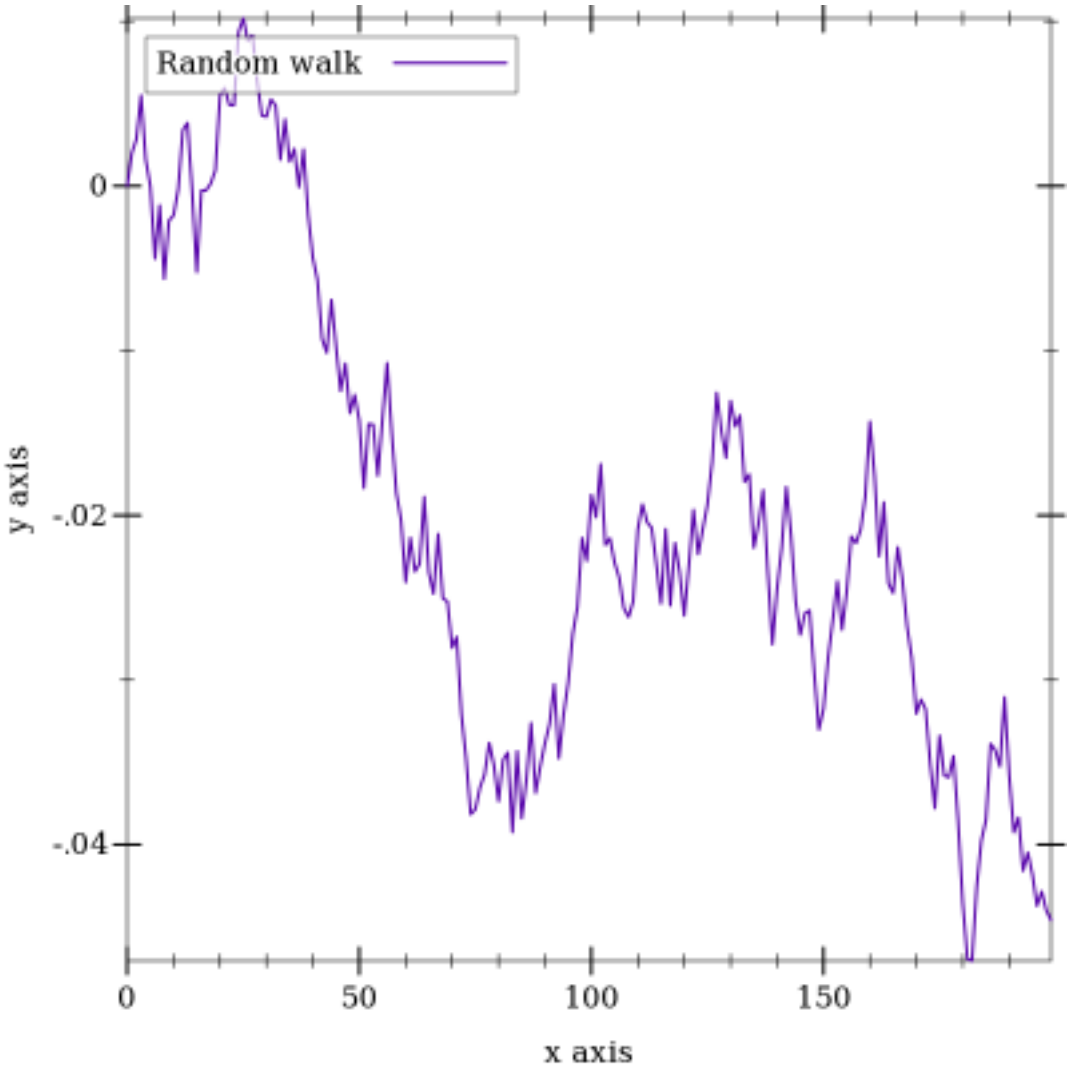



```
(lines  vs
        [#:x-min x-min
         #:x-max x-max
         #:y-min y-min
         #:y-max y-max
         #:color color
         #:width width
         #:style style
         #:alpha alpha
         #:label label]) → renderer2d?
  vs : (listof (vector/c real? real?))
```

```
  x-min : (or/c rational? #f) = #f
  x-max : (or/c rational? #f) = #f
  y-min : (or/c rational? #f) = #f
  y-max : (or/c rational? #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"))
```

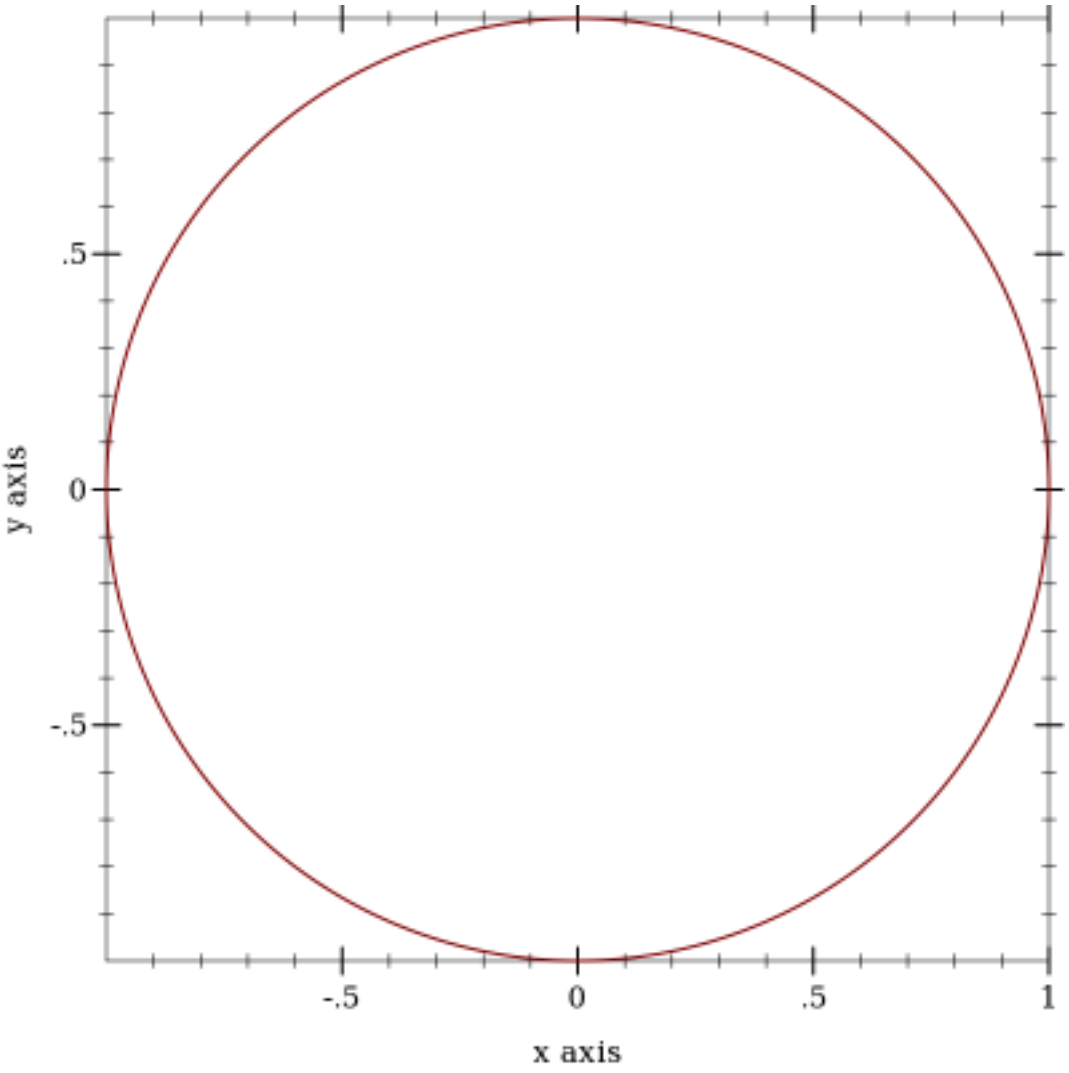


The `parametric` and `polar` functions are defined using `lines`.

```
(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 : rational?
  t-max : rational?
  x-min : (or/c rational? #f) = #f
  x-max : (or/c rational? #f) = #f
  y-min : (or/c rational? #f) = #f
  y-max : (or/c rational? #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)))
```

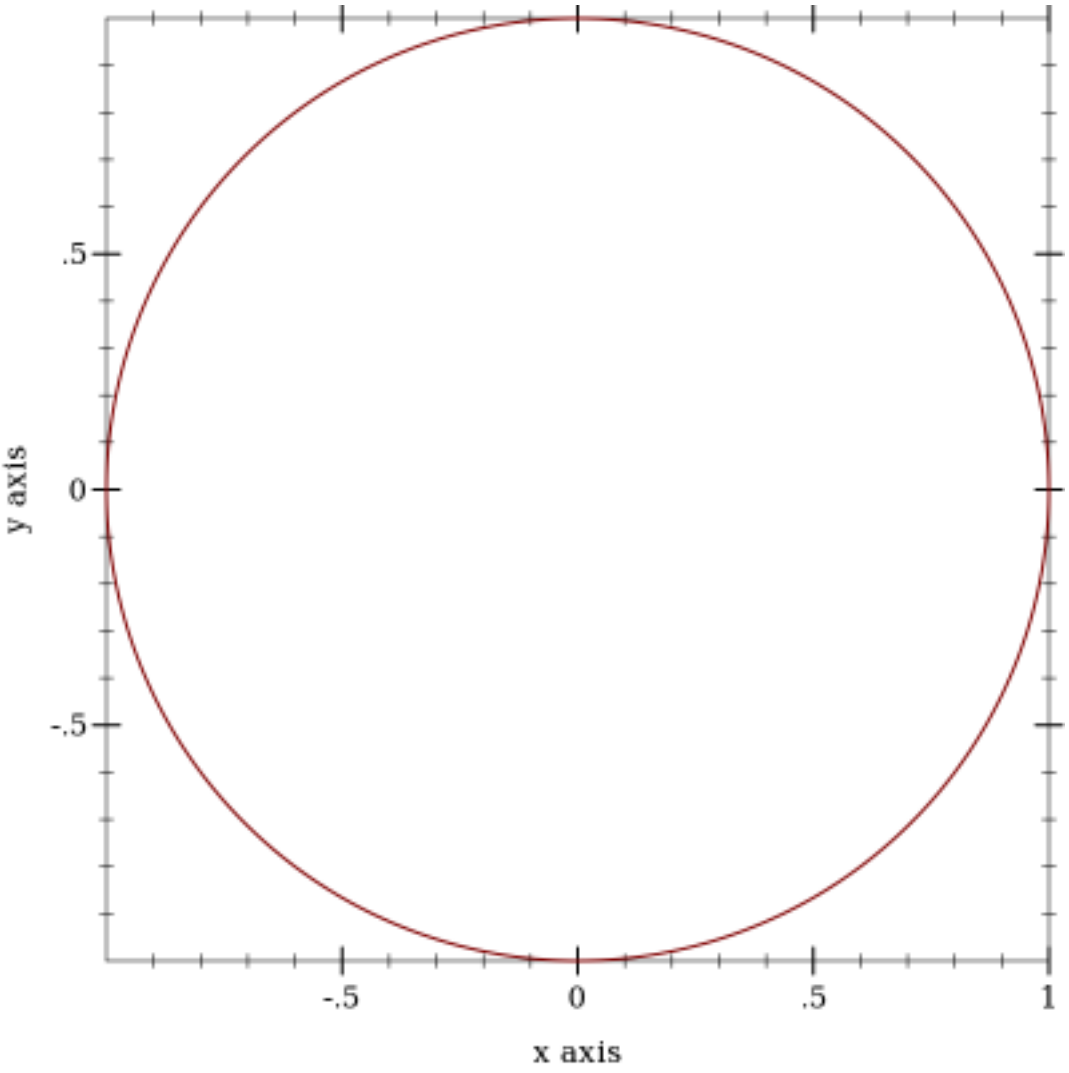

.5
0
-.5
y axis
-.5
0
.5
1
x axis

```
(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 rational? #f) = #f
  x-max : (or/c rational? #f) = #f
  y-min : (or/c rational? #f) = #f
  y-max : (or/c rational? #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)))
```

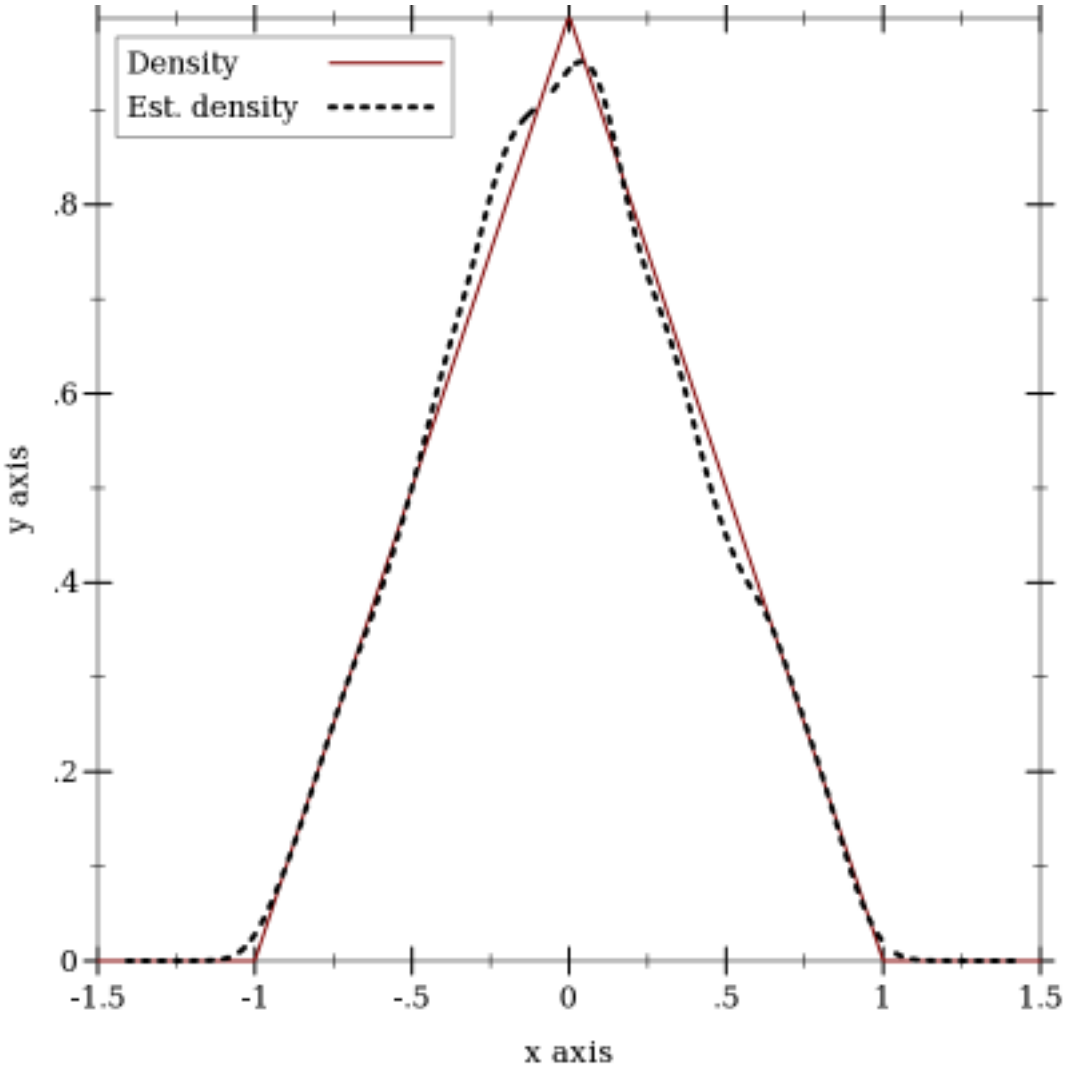

.5
0
-.5
y axis
-.5
0
.5
1
x axis

```
(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 rational? #f) = #f
  x-max : (or/c rational? #f) = #f
  y-min : (or/c rational? #f) = #f
  y-max : (or/c rational? #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")))
```



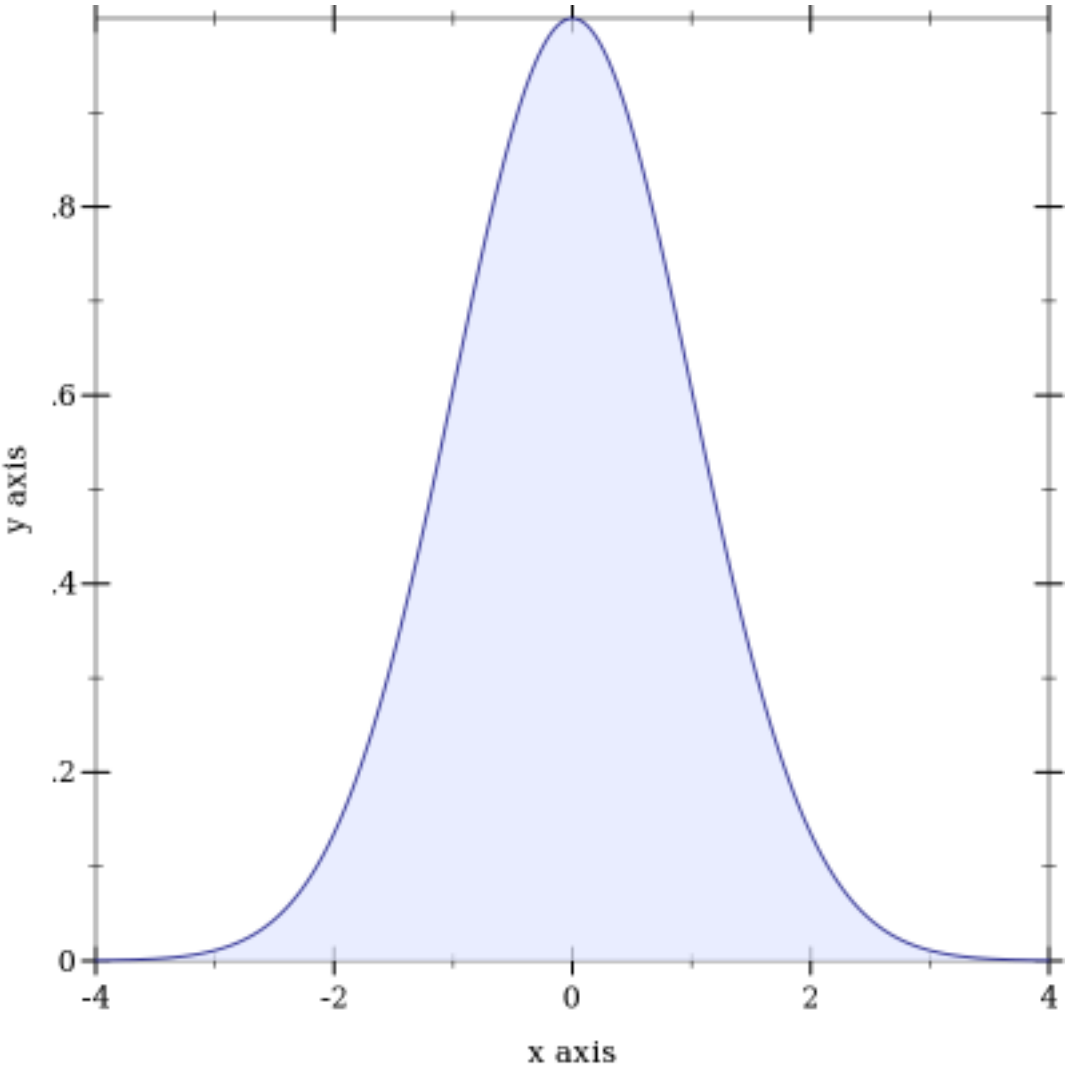
3.4 2D Interval Renderers

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 rational? #f) = #f
  x-max : (or/c rational? #f) = #f
  y-min : (or/c rational? #f) = #f
  y-max : (or/c rational? #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`.

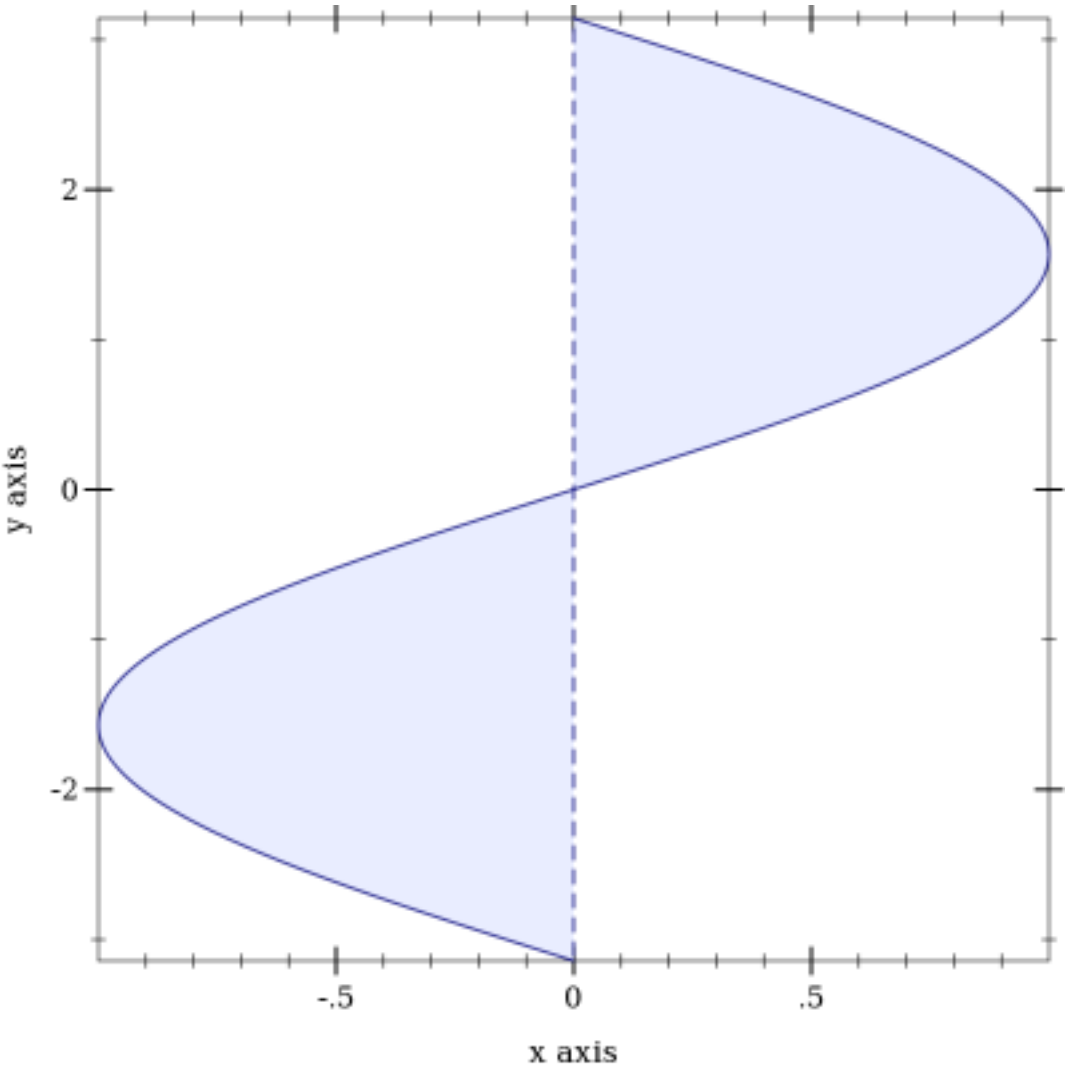
```
> (plot (function-interval (λ (x) 0) (λ (x) (exp (* -1/2 (sqr x))))
                           -4 4 #:line1-style 'transparent))
```


.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 rational? #f) = #f
  y-max : (or/c rational? #f) = #f
  x-min : (or/c rational? #f) = #f
  x-max : (or/c rational? #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`.

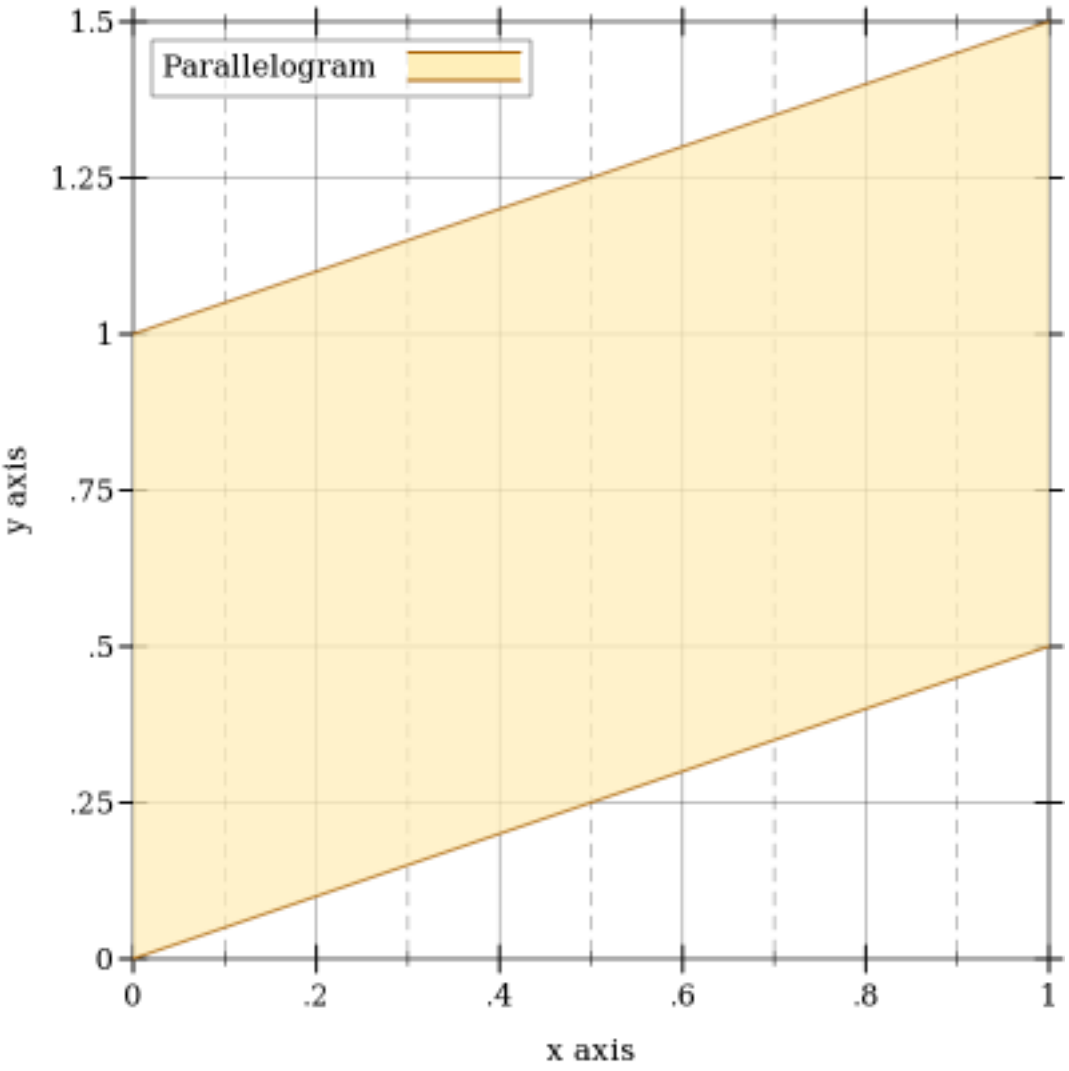
```
> (plot (inverse-interval sin (λ (x) 0) (- pi) pi
                          #:line2-style 'long-dash))
```


2
0
-2
y axis
-.5
0
.5
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 rational? #f) = #f
  x-max : (or/c rational? #f) = #f
  y-min : (or/c rational? #f) = #f
  y-max : (or/c rational? #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")))
```

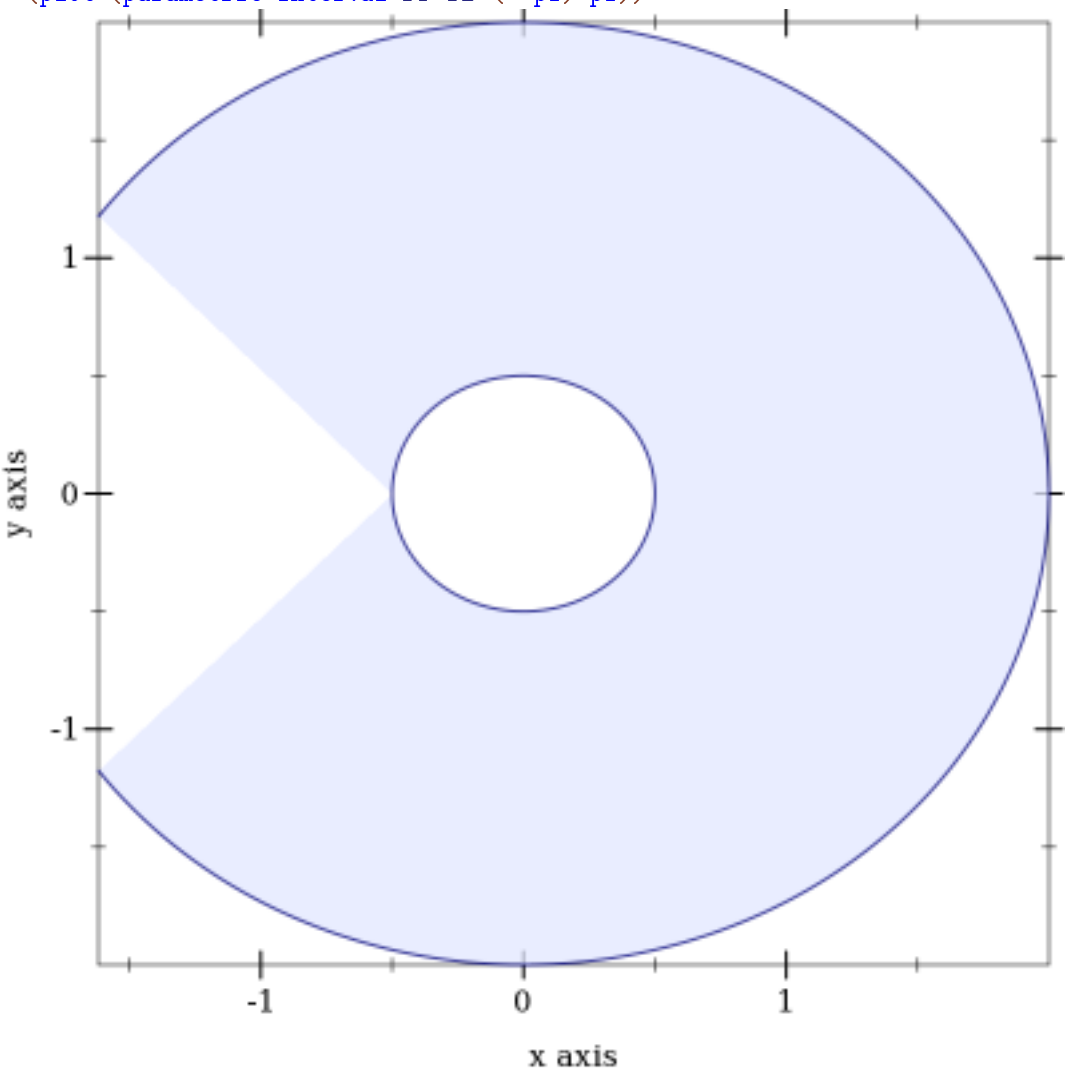

Parallelogram
1.5
1.25
1
.75
.5
.25
0
y axis
0
.2
.4
.6
.8
1
x axis

```
(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 : rational?
  t-max : rational?
  x-min : (or/c rational? #f) = #f
  x-max : (or/c rational? #f) = #f
  y-min : (or/c rational? #f) = #f
  y-max : (or/c rational? #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`.

```
> (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))
```

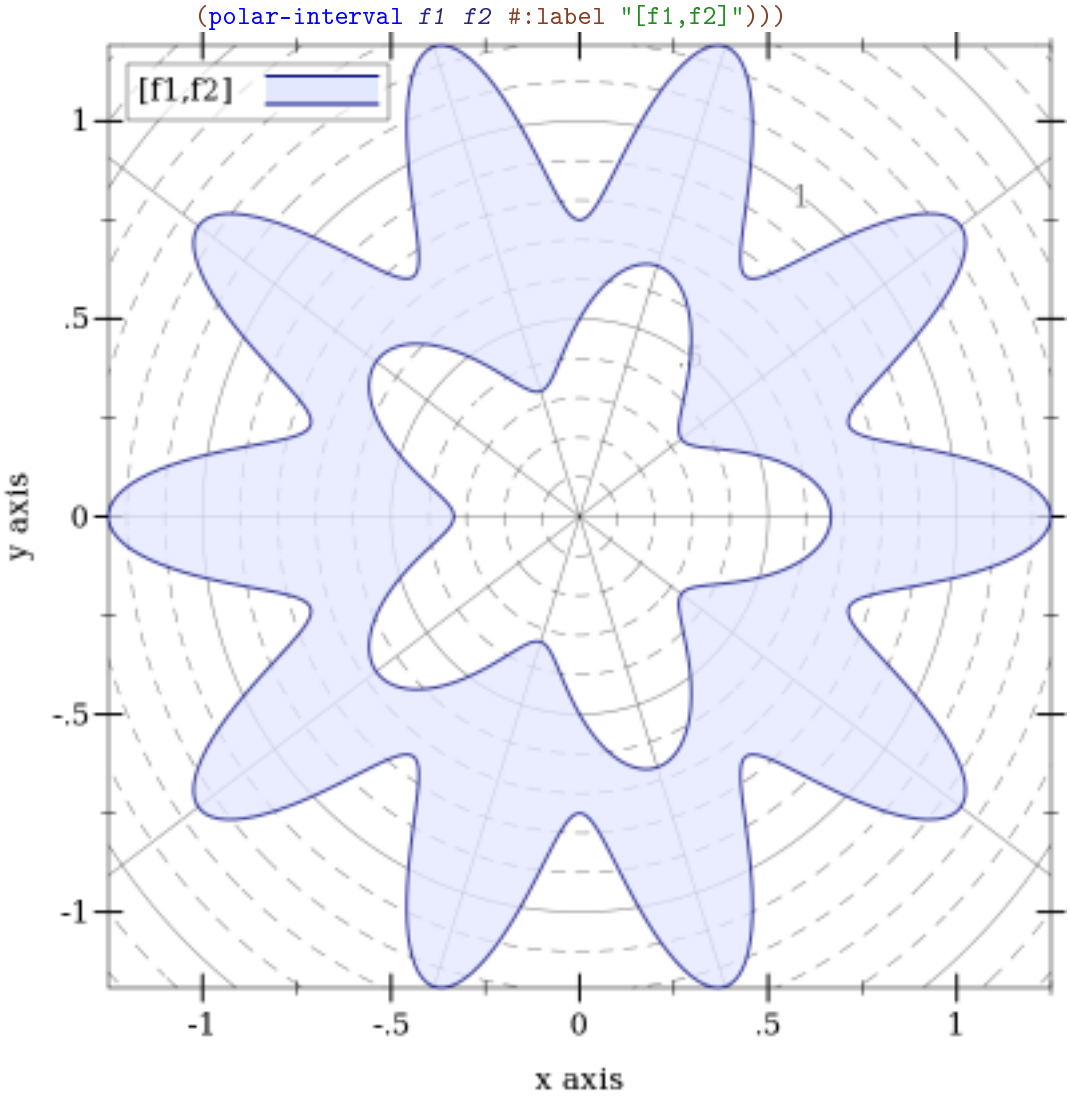


```
(polar-interval  f1
                 f2
                [θ-min
                 θ-max
                 #:x-min x-min
                 #:x-max x-max
                 #:y-min y-min
                 #:y-max y-max
                 #:samples samples
                 #:color color
                 #:style 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 : rational? = 0
  θ-max : rational? = (* 2 pi)
  x-min : (or/c rational? #f) = #f
  x-max : (or/c rational? #f) = #f
  y-min : (or/c rational? #f) = #f
  y-max : (or/c rational? #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`.

```
> (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]")))
```

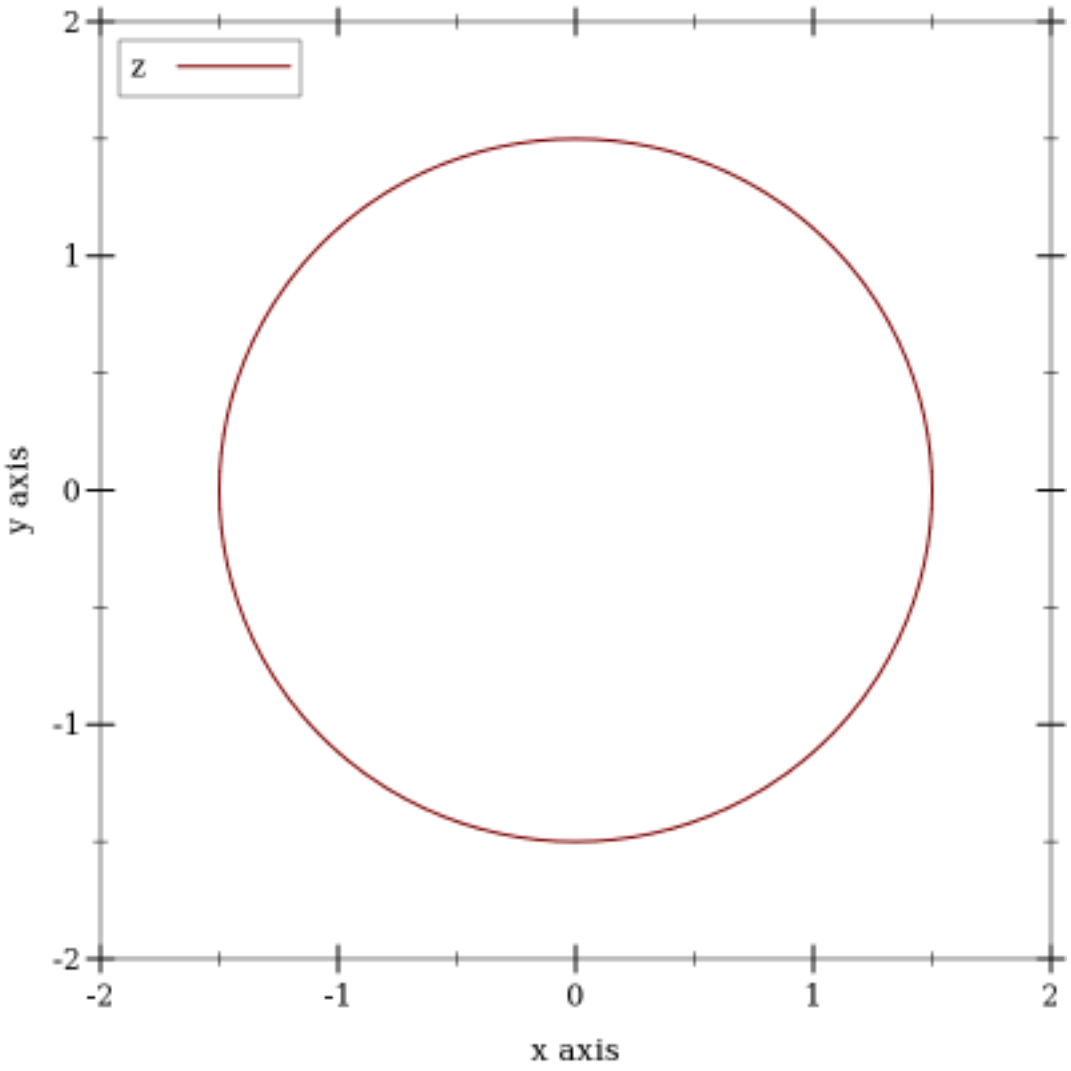


3.5 2D Contour (Isoline) Renderers

```
(isoline  f
          z
         [x-min
          x-max
          y-min
          y-max
          #:samples samples
          #:color color
          #:width width
          #:style style
          #:alpha alpha
          #:label label])    → renderer2d?
  f : (real? real? . -> . real?)
  z : real?
  x-min : (or/c rational? #f) = #f
  x-max : (or/c rational? #f) = #f
  y-min : (or/c rational? #f) = #f
  y-max : (or/c rational? #f) = #f
  samples : (and/c exact-integer? (>=/c 2)) = (contour-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 contour line, or a line of constant value (height). A circle of radius `r`, for example, is the line of constant value `r` for the distance function:

```
> (plot (isoline (λ (x y) (sqrt (+ (sqr x) (sqr y)))) 1.5
                 -2 2 -2 2 #:label "z"))
```



In this case, `r` = `1.5`.

This function would have been named `contour`, except the name was already used by a deprecated function. It may be renamed in the future, with `isoline` as an alias.

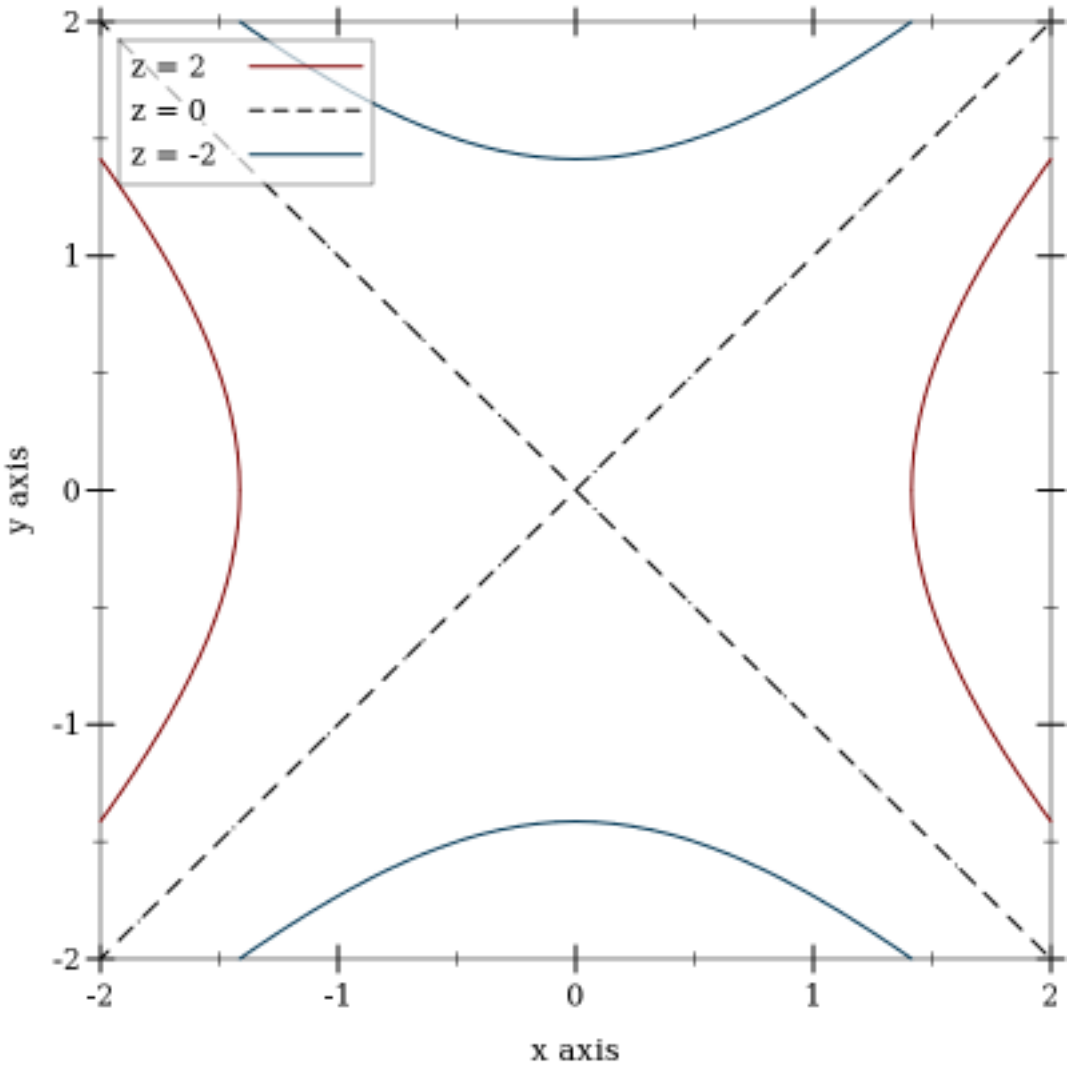
```
(contours  f
           [x-min
            x-max
            y-min
            y-max
            #:samples samples
            #:levels levels
            #:colors colors
            #:widths widths
            #:styles styles
            #:alphas alphas
            #:label label])    → renderer2d?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

Returns a renderer that plots contour lines, or lines of constant value (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"))
```

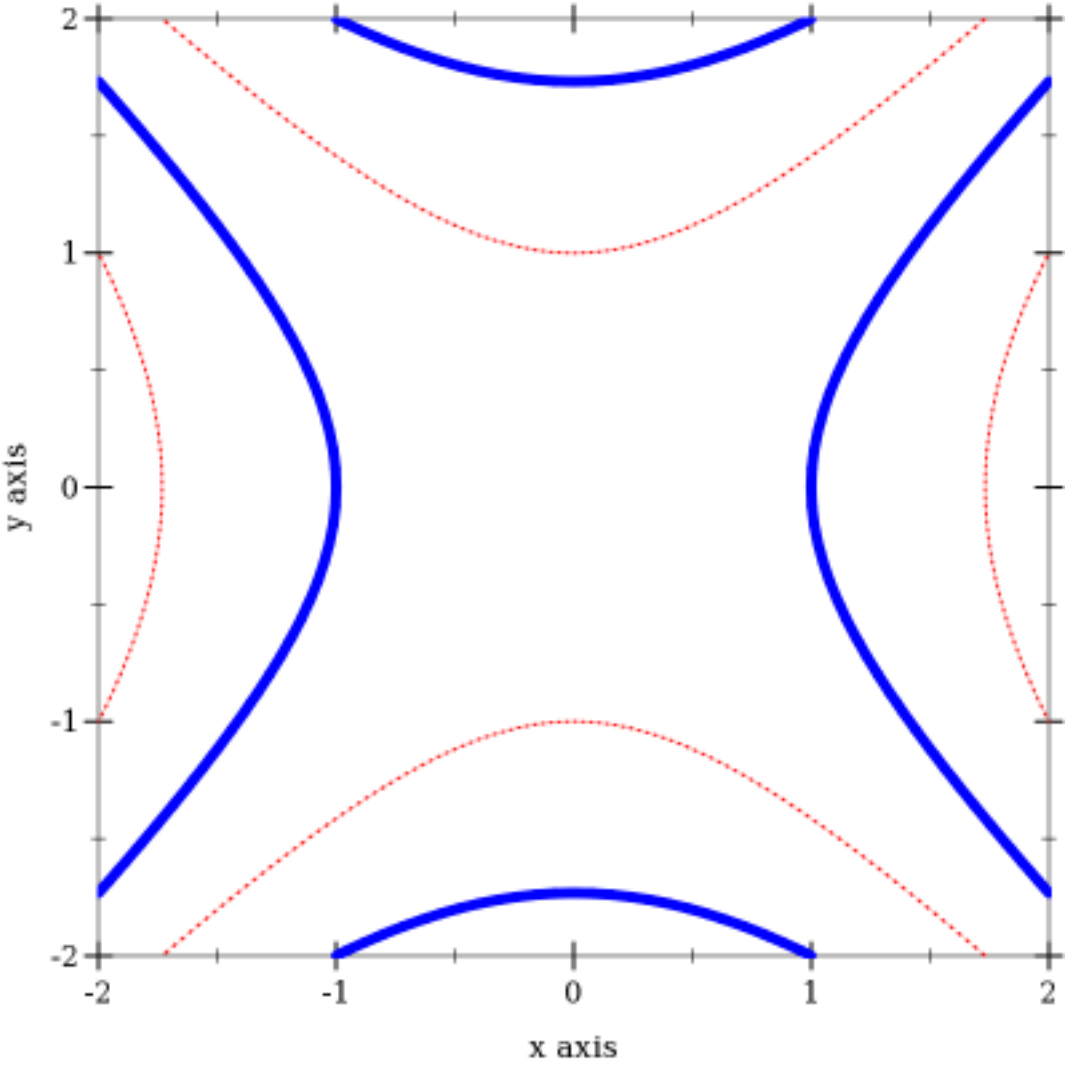


The appearance keyword arguments assign a color, width, style and opacity *to each contour line*. Each can be given as a list or as a function from a list of output values of `f` to a list of appearance values. In both cases, when there are more contour lines than list elements, the colors, widths, styles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

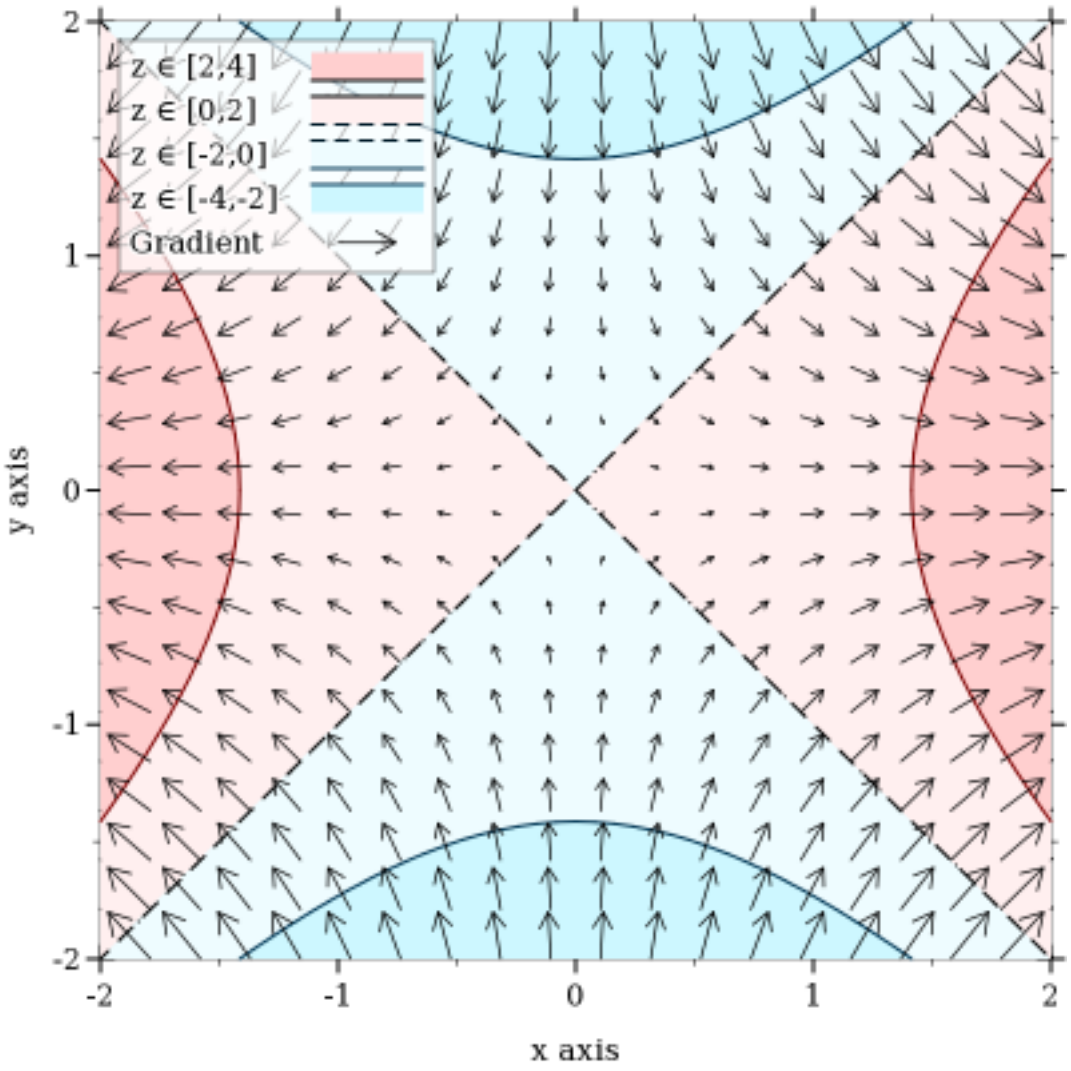


```
(contour-intervals  f
                   [x-min
                    x-max
                    y-min
                    y-max
                    #:samples samples
                    #:levels levels
                    #:colors colors
                    #:styles styles
                    #:contour-colors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

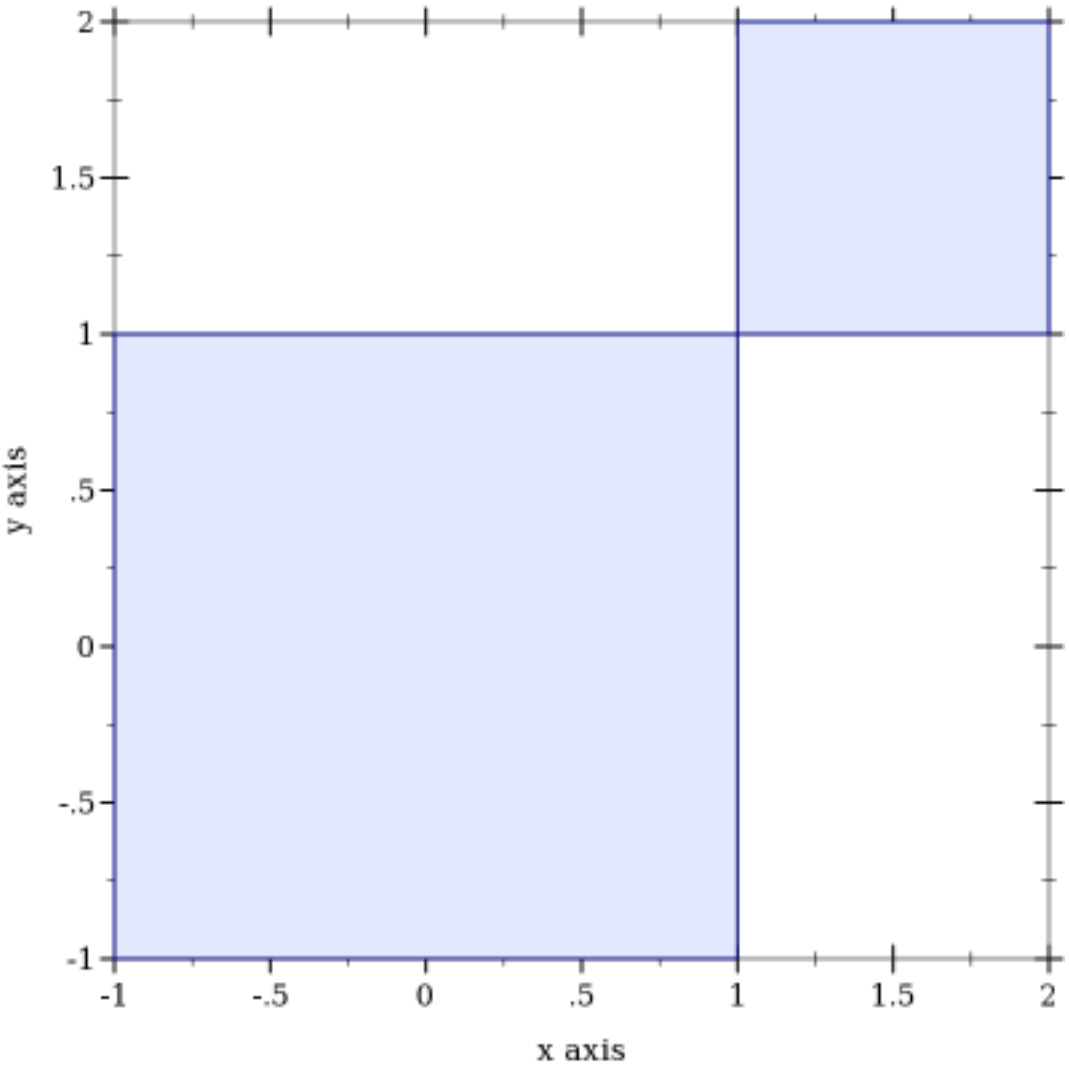


3.6 2D Rectangle Renderers

```
(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 rational? #f) = #f
  x-max : (or/c rational? #f) = #f
  y-min : (or/c rational? #f) = #f
  y-max : (or/c rational? #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,

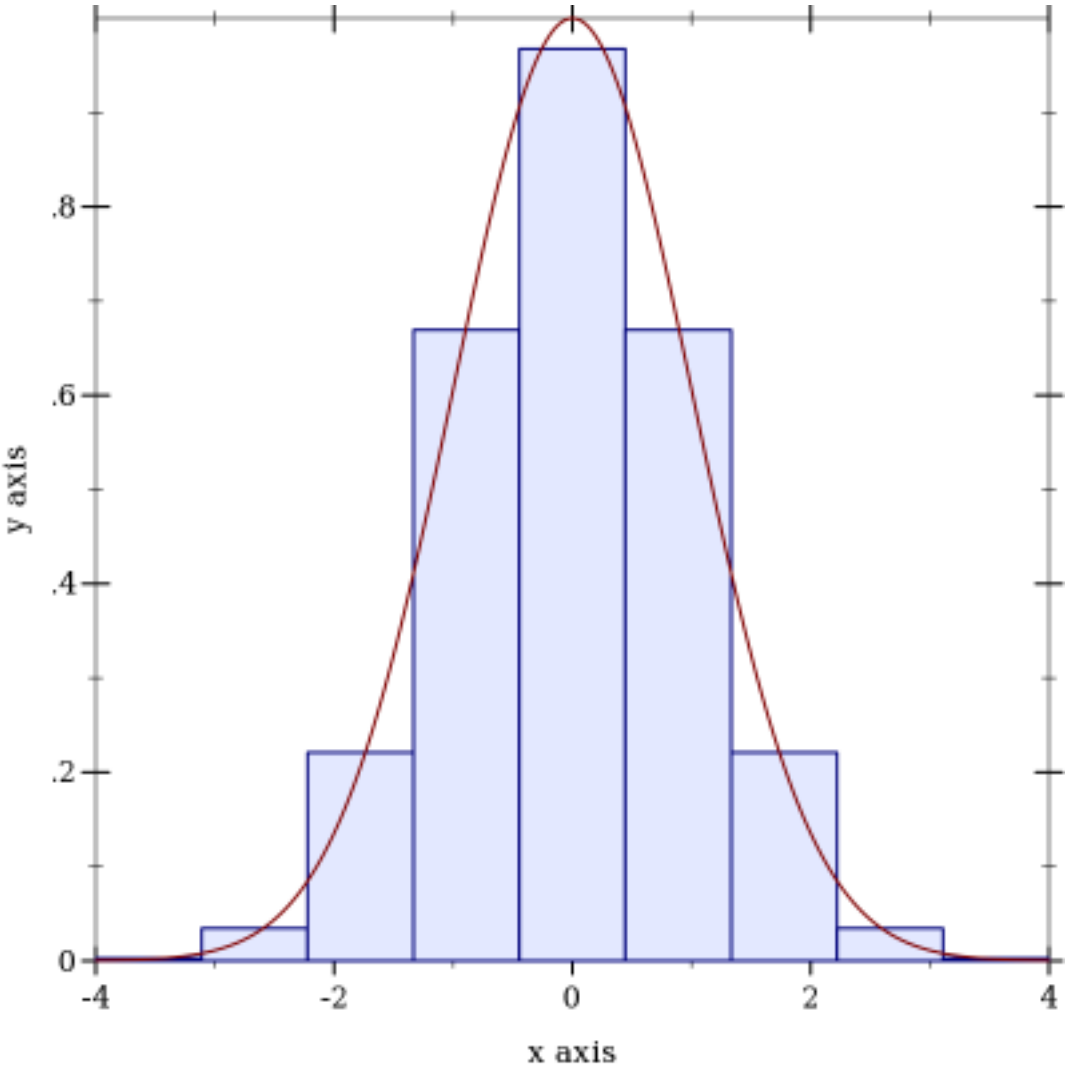
```
> (plot (rectangles (list (vector (ivl -1 1) (ivl -1 1))
                          (vector (ivl 1 2) (ivl 1 2)))))
```


2
1.5
1
.5
0
-.5
-1
y axis
-1
-.5
0
.5
1
1.5
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 rational? #f) = #f
  x-max : (or/c rational? #f) = #f
  y-min : (or/c rational? #f) = 0
  y-max : (or/c rational? #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.

```
> (require (only-in plot/utils linear-seq))

> (define (f x) (exp (* -1/2 (sqr x))))

> (plot (list (area-histogram f (linear-seq -4 4 10))
              (function f -4 4)))
```

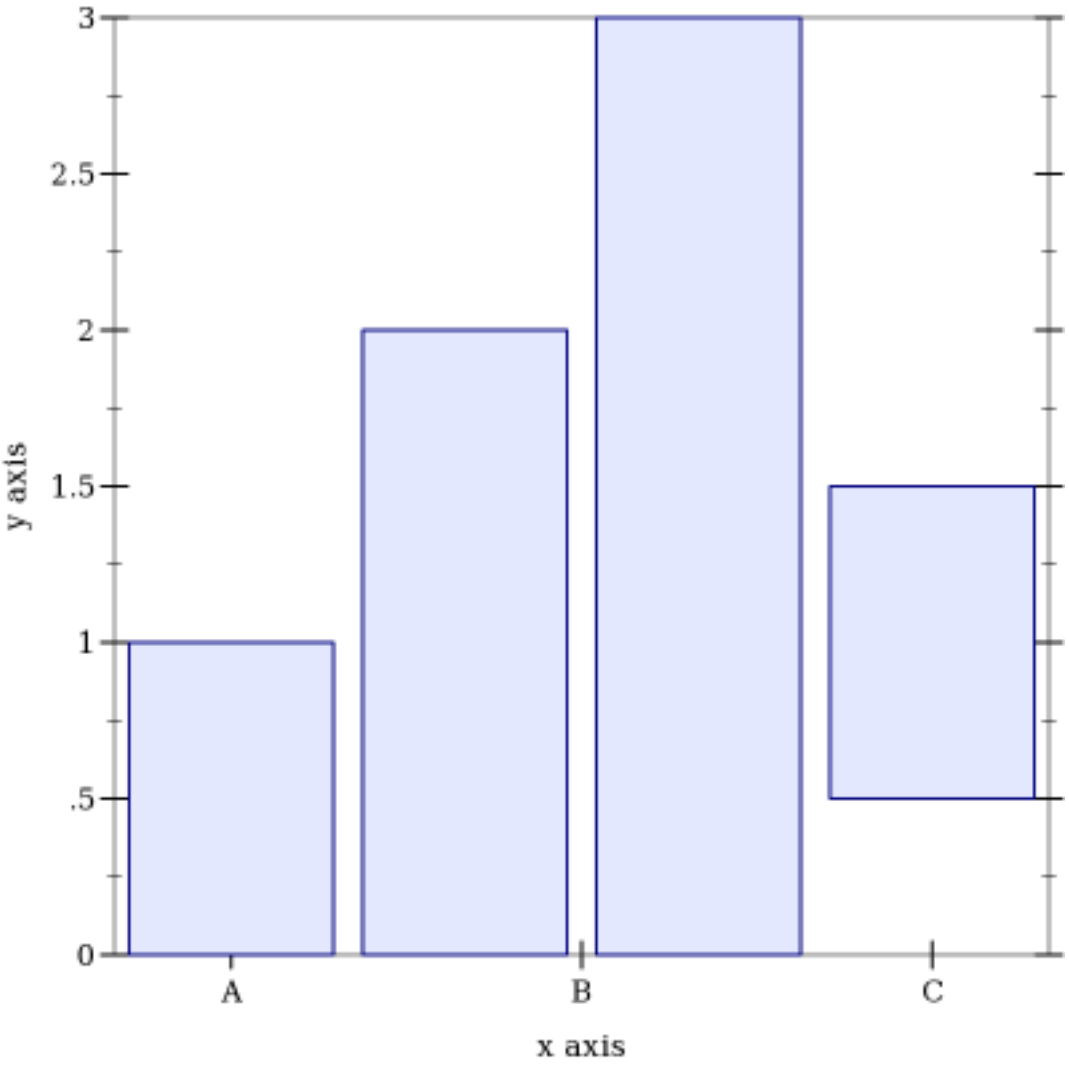

.8
.6
.4
.2
0
y axis
-4
-2
0
2
4
x axis

```
(discrete-histogram  cat-vals
                     [#:x-min x-min
                      #:x-max x-max
                      #:y-min y-min
                      #:y-max y-max
                      #:gap gap
                      #:skip skip
                      #:invert? invert?
                      #:color color
                      #:style style
                      #:line-color line-color
                      #:line-width line-width
                      #:line-style line-style
                      #:alpha alpha
                      #:label label
                      #:add-ticks? add-ticks?
                      #:far-ticks? far-ticks?]) → renderer2d?
  cat-vals : (listof (vector/c any/c (or/c real? ivl? #f)))
  x-min : (or/c rational? #f) = 0
  x-max : (or/c rational? #f) = #f
  y-min : (or/c rational? #f) = 0
  y-max : (or/c rational? #f) = #f
  gap : (real-in 0 1) = (discrete-histogram-gap)
  skip : (>=/c 0) = (discrete-histogram-skip)
  invert? : boolean? = (discrete-histogram-invert?)
  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
  add-ticks? : boolean? = #t
  far-ticks? : boolean? = #f
```

Returns a renderer that draws a discrete histogram.

Example:

```
> (plot (discrete-histogram (list #(A 1) #(B 2) #(B 3)
                                  (vector 'C (ivl 0.5 1.5)))))
```



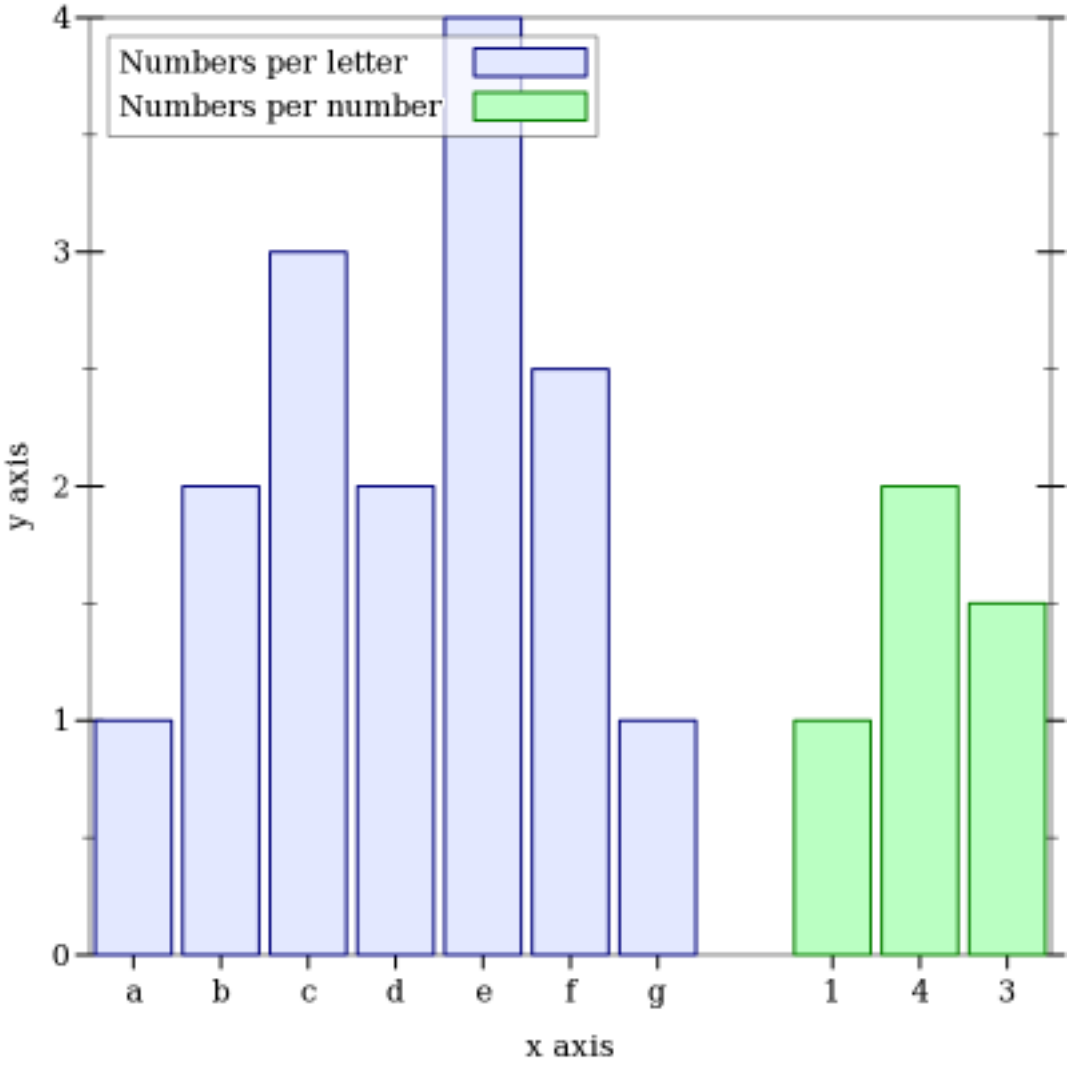
Use `#:invert? #t` to draw horizontal bars. See `stacked-histogram` for an example.

Each bar takes up exactly one plot unit, and is drawn with `(* 1/2 gap)` empty space on each side. Bar number `i` is drawn at `(+ x-min (* i skip))`. Thus, the first bar (`i` = `0`) is drawn in the interval between `x-min` (default `0`) and `(+ x-min 1)`.

To plot two 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))
```

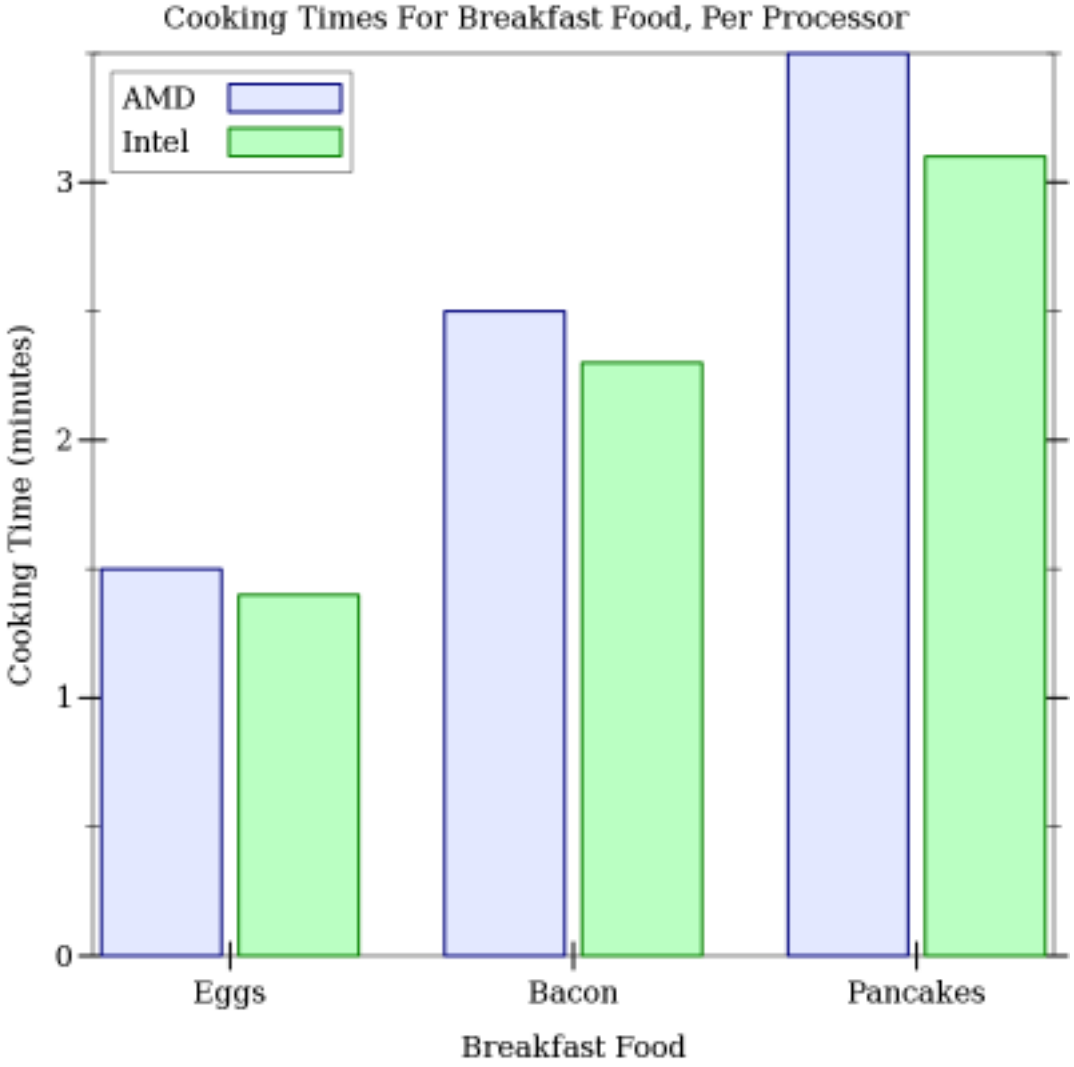
```
                        #:label "Numbers per letter")
(discrete-histogram (list #(1 1) #(4 2) #(3 1.5))
                    #:x-min 8
                    #:label "Numbers per number"
                    #:color 2 #:line-color 2)))
```



Here, the first histogram has 7 bars, so the second is drawn starting at *x-min* = 8.

To interleave histograms, such as when plotting benchmark results, use a *skip* value larger than or equal to the number of histograms, and give each histogram a different *x-min*. For example,

```
> (plot (list (discrete-histogram
               '(#(Eggs 1.5) #(Bacon 2.5) #(Pancakes 3.5))
               #:skip 2.5 #:x-min 0
               #:label "AMD")
              (discrete-histogram
               '(#(Eggs 1.4) #(Bacon 2.3) #(Pancakes 3.1))
               #:skip 2.5 #:x-min 1
               #:label "Intel" #:color 2 #:line-color 2))
        #:x-label "Breakfast Food" #:y-label "Cooking Time (min-
utes)"
        #:title "Cooking Times For Breakfast Food, Per Processor")
```



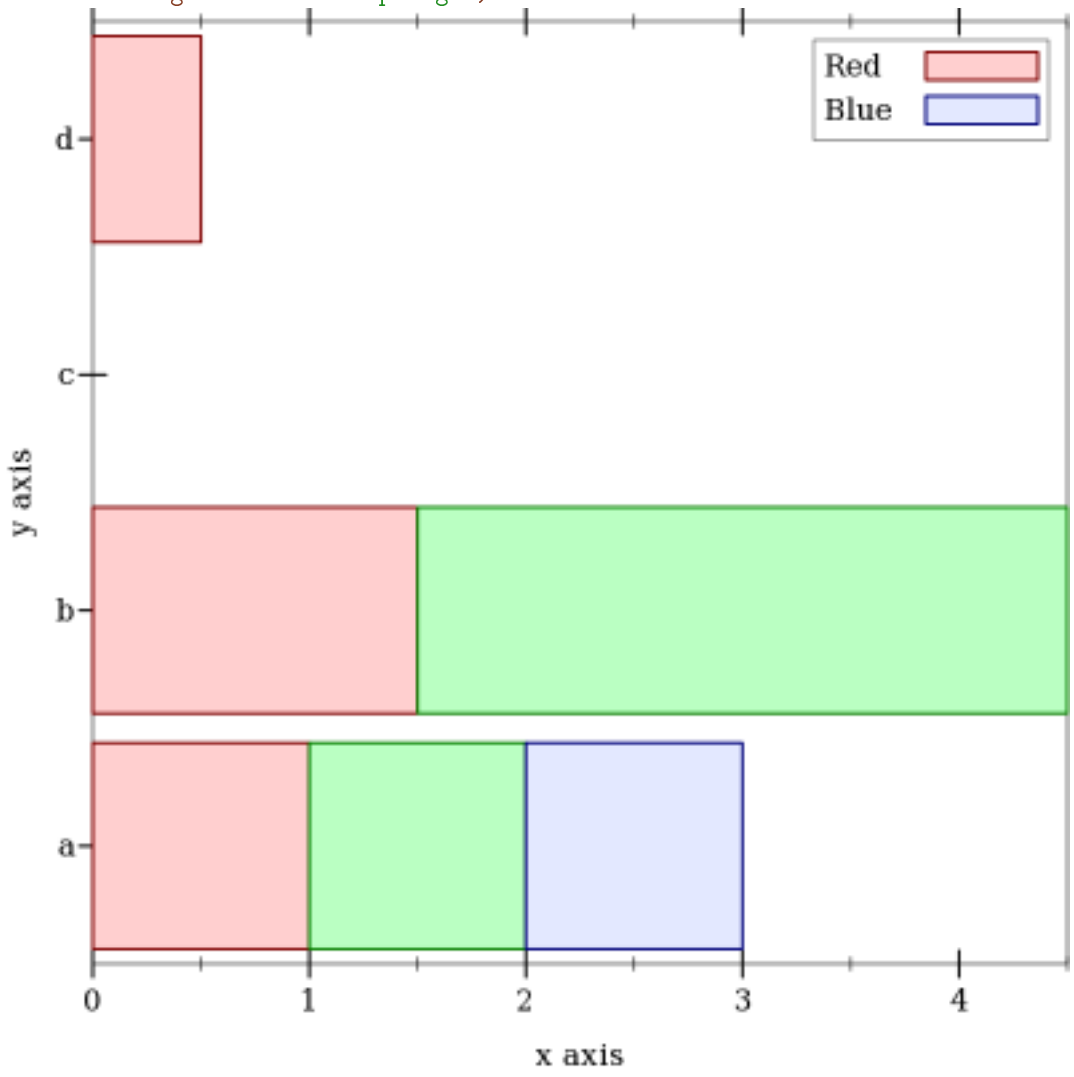
When interleaving many histograms, consider setting the discrete-histogram-skip parameter to change *skip*'s default value.

```
(stacked-histogram  cat-vals
                    [#:x-min x-min
                     #:x-max x-max
                     #:y-min y-min
                     #:y-max y-max
                     #:gap gap
                     #:skip skip
                     #:invert? invert?
                     #:colors colors
                     #:styles styles
                     #:line-colors line-colors
                     #:line-widths line-widths
                     #:line-styles line-styles
                     #:alphas alphas
                     #:labels labels
                     #:add-ticks? add-ticks?
                     #:far-ticks? far-ticks?])
 → (listof renderer2d?)
  cat-vals : (listof (vector/c any/c (listof real?)))
  x-min : (or/c rational? #f) = 0
  x-max : (or/c rational? #f) = #f
  y-min : (or/c rational? #f) = 0
  y-max : (or/c rational? #f) = #f
  gap : (real-in 0 1) = (discrete-histogram-gap)
  skip : (>=/c 0) = (discrete-histogram-skip)
  invert? : boolean? = (discrete-histogram-invert?)
  colors : (plot-colors/c nat/c) = (stacked-histogram-colors)
  styles : (plot-brush-styles/c nat/c)
         = (stacked-histogram-styles)
  line-colors : (plot-colors/c nat/c)
              = (stacked-histogram-line-colors)
  line-widths : (pen-widths/c nat/c)
              = (stacked-histogram-line-widths)
  line-styles : (plot-pen-styles/c nat/c)
              = (stacked-histogram-line-styles)
  alphas : (alphas/c nat/c) = (stacked-histogram-alphas)
  labels : (labels/c nat/c) = '(#f)
  add-ticks? : boolean? = #t
  far-ticks? : boolean? = #f
```

Returns a list of renderers that draw parts of a stacked histogram. The heights of each bar section are given as a list.

Example:

```
> (plot (stacked-histogram (list #(a (1 1 1)) #(b (1.5 3))
                                 #(c ()) #(d (1/2)))
                           #:invert? #t
                           #:labels '("Red" #f "Blue"))
        #:legend-anchor 'top-right)
```



3.7 2D Plot Decoration Renderers

```
(x-axis [y
         #:ticks? ticks?
         #:labels? labels?
         #:far? far?
         #:alpha alpha])   → renderer2d?
  y : real? = 0
  ticks? : boolean? = (x-axis-ticks?)
  labels? : boolean? = (x-axis-labels?)
  far? : boolean? = (x-axis-far?)
  alpha : (real-in 0 1) = (x-axis-alpha)
```

Returns a renderer that draws an x axis.

```
(y-axis [x
         #:ticks? ticks?
         #:labels? labels?
         #:far? far?
         #:alpha alpha])   → renderer2d?
  x : real? = 0
  ticks? : boolean? = (y-axis-ticks?)
  labels? : boolean? = (y-axis-labels?)
  far? : boolean? = (y-axis-far?)
  alpha : (real-in 0 1) = (y-axis-alpha)
```

Returns a renderer that draws a y axis.

```
(axes [x
       y
       #:x-ticks? x-ticks?
       #:y-ticks? y-ticks?
       #:x-labels? x-labels?
       #:y-labels? y-labels?
       #:x-alpha x-alpha
       #:y-alpha y-alpha])   → (listof renderer2d?)
  x : real? = 0
  y : real? = 0
  x-ticks? : boolean? = (x-axis-ticks?)
  y-ticks? : boolean? = (y-axis-ticks?)
  x-labels? : boolean? = (x-axis-labels?)
  y-labels? : boolean? = (y-axis-labels?)
  x-alpha : (real-in 0 1) = (x-axis-alpha)
  y-alpha : (real-in 0 1) = (y-axis-alpha)
```

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

```
(polar-axes [#:number num
             #:ticks? ticks?
             #:labels? labels?
             #:alpha alpha])   → renderer2d?
  num : exact-nonnegative-integer? = (polar-axes-number)
  ticks? : boolean? = (polar-axes-ticks?)
  labels? : boolean? = (polar-axes-labels?)
  alpha : (real-in 0 1) = (polar-axes-alpha)
```

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?)
```

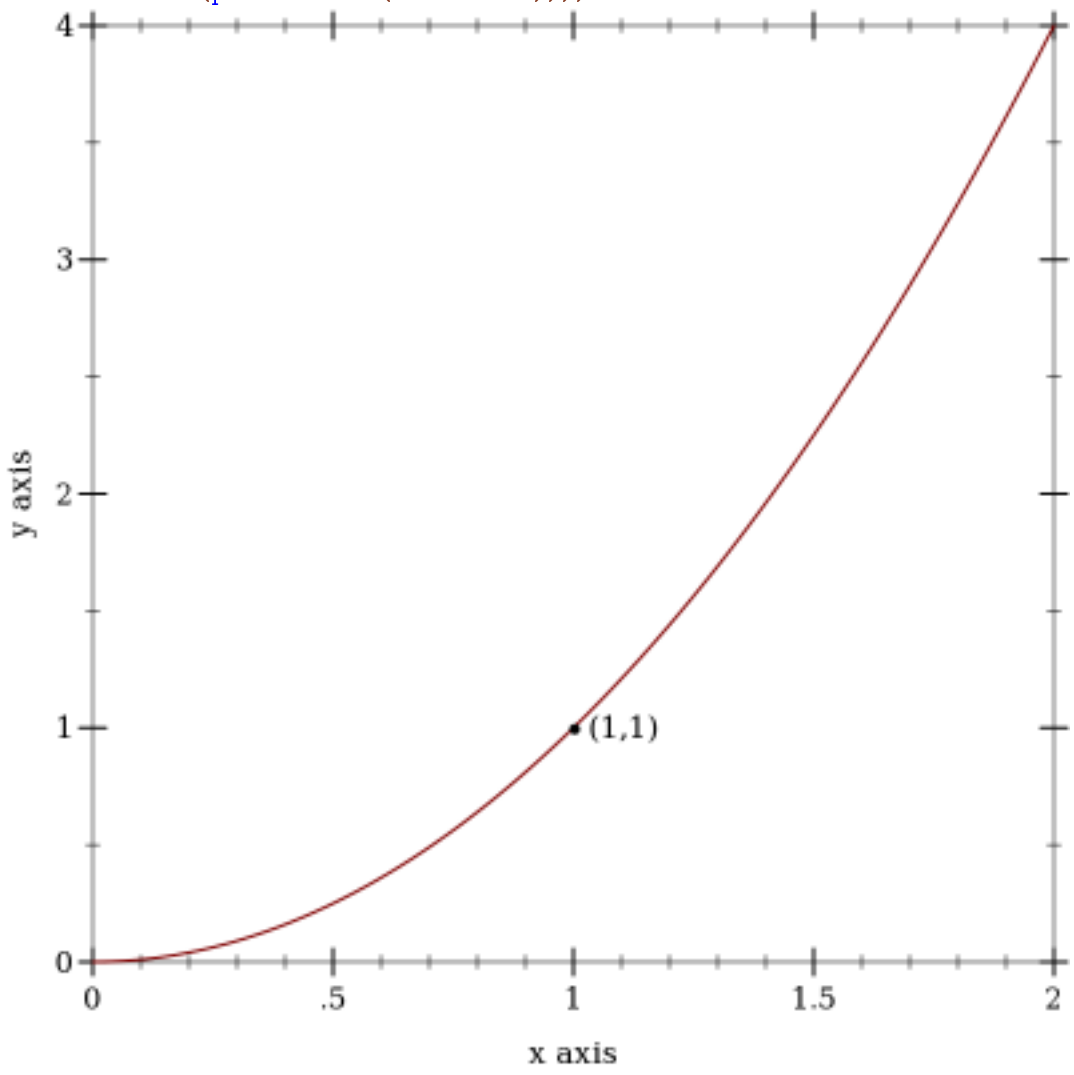
Returns a list containing an `x-tick-lines` renderer and a `y-tick-lines` renderer. See `lines-interval` for an example.

```
(point-label  v
              [label
               #:color color
               #:size size
               #:family family
               #:anchor anchor
               #:angle angle
               #:point-color point-color
               #:point-fill-color point-fill-color
               #:point-size point-size
               #:point-line-width point-line-width
               #:point-sym point-sym
               #: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)
  family : font-family/c = (plot-font-family)
  anchor : anchor/c = (label-anchor)
  angle : real? = (label-angle)
```

```
point-color : plot-color/c = (point-color)
point-fill-color : (or/c plot-color/c 'auto) = 'auto
point-size : (>=/c 0) = (label-point-size)
point-line-width : (>=/c 0) = (point-line-width)
point-sym : point-sym/c = 'fullcircle
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))))
```

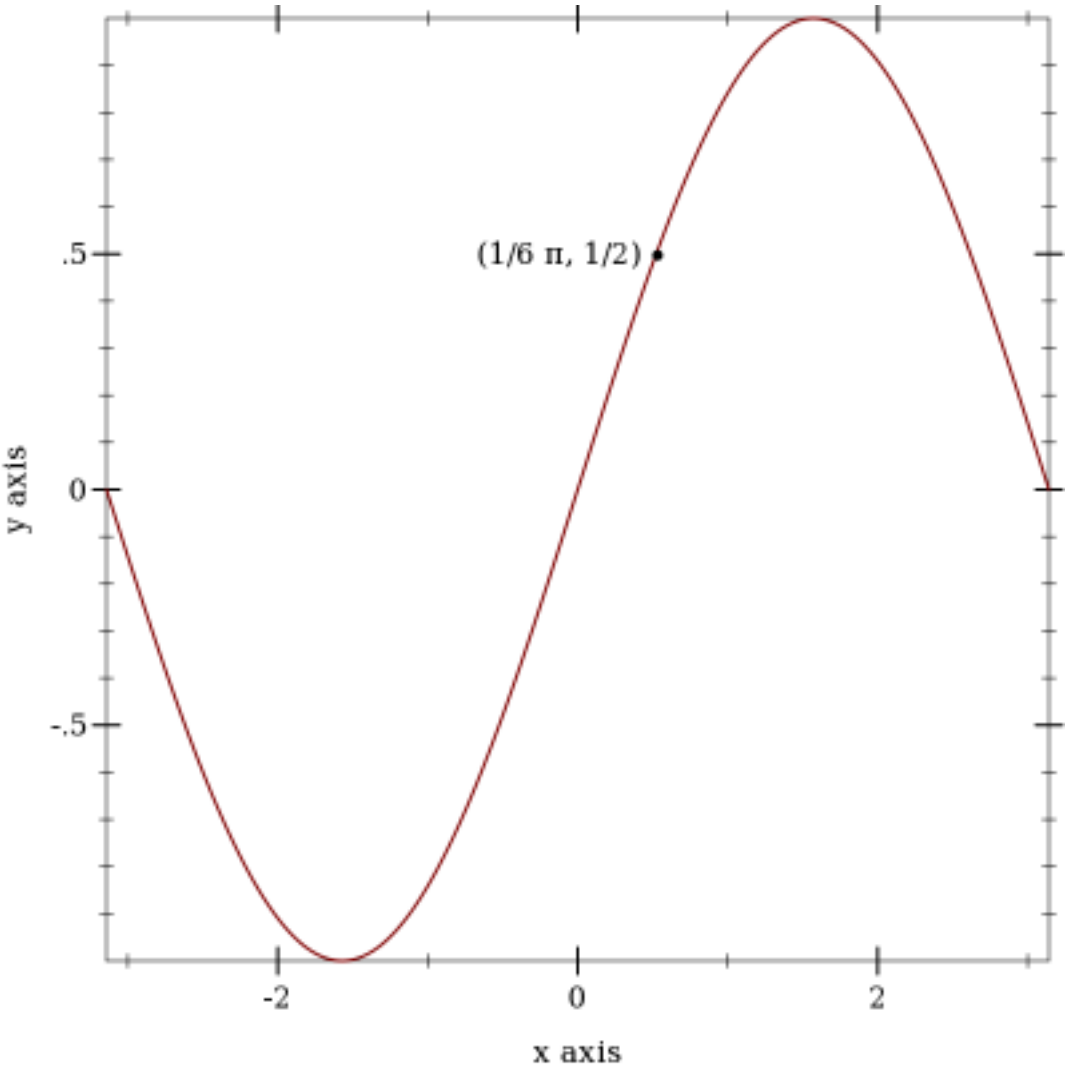


The remaining labeled-point functions are defined in terms of this one.

```
(function-label  f
                 x
                [label
                 #:color color
                 #:size size
                 #:family family
                 #:anchor anchor
                 #:angle angle
                 #:point-color point-color
                 #:point-fill-color point-fill-color
                 #:point-size point-size
                 #:point-line-width point-line-width
                 #:point-sym point-sym
                 #: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)
  family : font-family/c = (plot-font-family)
  anchor : anchor/c = (label-anchor)
  angle : real? = (label-angle)
  point-color : plot-color/c = (point-color)
  point-fill-color : (or/c plot-color/c 'auto) = 'auto
  point-size : (>=/c 0) = (label-point-size)
  point-line-width : (>=/c 0) = (point-line-width)
  point-sym : point-sym/c = 'fullcircle
  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)))
```


(1/6 π, 1/2)
.5
0
-.5
y axis
-2
0
2
x axis

```
(inverse-label  f
                y
               [label
                #:color color
                #:size size
                #:family family
                #:anchor anchor
                #:angle angle
                #:point-color point-color
                #:point-fill-color point-fill-color
                #:point-size point-size
                #:point-line-width point-line-width
                #:point-sym point-sym
                #: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)
  family : font-family/c = (plot-font-family)
  anchor : anchor/c = (label-anchor)
  angle : real? = (label-angle)
  point-color : plot-color/c = (point-color)
  point-fill-color : (or/c plot-color/c 'auto) = 'auto
  point-size : (>=/c 0) = (label-point-size)
  point-line-width : (>=/c 0) = (point-line-width)
  point-sym : point-sym/c = 'fullcircle
  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
                   #:family family
                   #:anchor anchor
                   #:angle angle
                   #:point-color point-color
                   #:point-fill-color point-fill-color
                   #:point-size point-size
                   #:point-line-width point-line-width
                   #:point-sym point-sym
                   #: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)
 family : font-family/c = (plot-font-family)
 anchor : anchor/c = (label-anchor)
 angle : real? = (label-angle)
 point-color : plot-color/c = (point-color)
 point-fill-color : (or/c plot-color/c 'auto) = 'auto
 point-size : (>=/c 0) = (label-point-size)
 point-line-width : (>=/c 0) = (point-line-width)
 point-sym : point-sym/c = 'fullcircle
 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
              #:family family
              #:anchor anchor
              #:angle angle
              #:point-color point-color
              #:point-fill-color point-fill-color
              #:point-size point-size
              #:point-line-width point-line-width
              #:point-sym point-sym
              #: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)
 family : font-family/c = (plot-font-family)
 anchor : anchor/c = (label-anchor)
 angle : real? = (label-angle)
 point-color : plot-color/c = (point-color)
 point-fill-color : (or/c plot-color/c 'auto) = 'auto
 point-size : (>=/c 0) = (label-point-size)
 point-line-width : (>=/c 0) = (point-line-width)
 point-sym : point-sym/c = 'fullcircle
 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 (or/c renderer3d? nonrenderer?))
  x-min : (or/c rational? #f) = #f
  x-max : (or/c rational? #f) = #f
  y-min : (or/c rational? #f) = #f
  y-max : (or/c rational? #f) = #f
  z-min : (or/c rational? #f) = #f
  z-max : (or/c rational? #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 (or/c renderer3d? nonrenderer?))
  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 (or/c renderer3d? nonrenderer?))
(plot3d-bitmap renderer-tree ...) → (is-a?/c bitmap%)
  renderer-tree : (treeof (or/c renderer3d? nonrenderer?))
(plot3d-snip renderer-tree ...) → (is-a?/c image-snip%)
  renderer-tree : (treeof (or/c renderer3d? nonrenderer?))
(plot3d-frame renderer-tree ...) → (is-a?/c frame%)
  renderer-tree : (treeof (or/c renderer3d? nonrenderer?))
```

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 (or/c renderer3d? nonrenderer?))
  dc : (is-a?/c dc<%>)
  x : real?
  y : real?
  width : (>=/c 0)
  height : (>=/c 0)
  x-min : (or/c rational? #f) = #f
  x-max : (or/c rational? #f) = #f
  y-min : (or/c rational? #f) = #f
  y-max : (or/c rational? #f) = #f
  z-min : (or/c rational? #f) = #f
  z-max : (or/c rational? #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

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

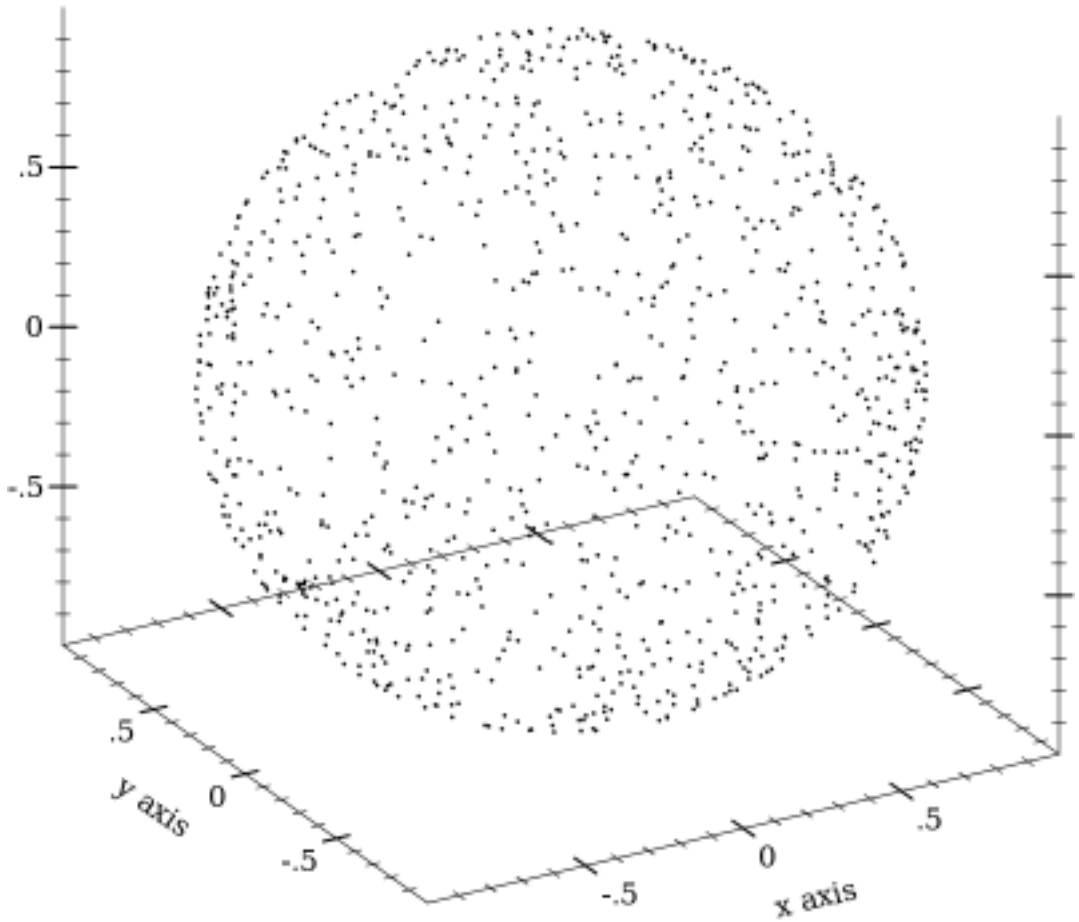
```
(points3d  vs
           [#:x-min x-min
            #:x-max x-max
            #:y-min y-min
            #:y-max y-max
            #:z-min z-min
            #:z-max z-max
            #:sym sym
            #:color color
            #:fill-color fill-color
            #:size size
            #:line-width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

```
color : plot-color/c = (point-color)
fill-color : (or/c plot-color/c 'auto) = 'auto
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:

```
> (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)
```

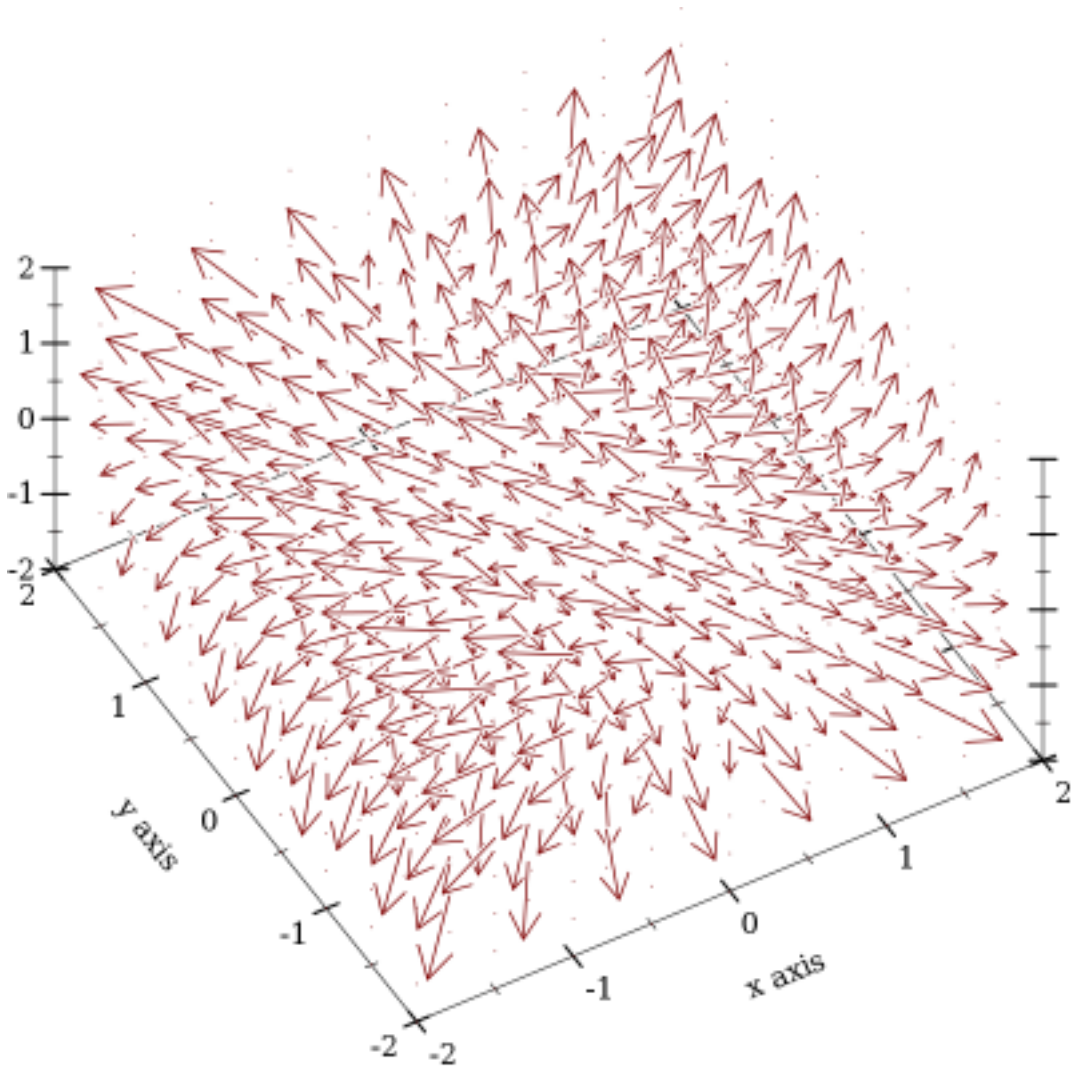

.5
0
-.5
.5
0
-.5
y axis
-.5
0
.5
x axis

```
(vector-field3d  f
                [x-min
                 x-max
                 y-min
                 y-max
                 z-min
                 z-max
                 #:samples samples
                 #:scale scale
                 #:color color
                 #:line-width line-width
                 #:line-style line-style
                 #:alpha alpha
                 #:label label])          → renderer3d?
     (or/c (real? real? real? . -> . (vector/c real? real? real?))
  f :       ((vector/c real? real? real?)
             . -> . (vector/c real? real? real?)))
  x-min : (or/c rational? #f) = #f
  x-max : (or/c rational? #f) = #f
  y-min : (or/c rational? #f) = #f
  y-max : (or/c rational? #f) = #f
  z-min : (or/c rational? #f) = #f
  z-max : (or/c rational? #f) = #f
  samples : exact-positive-integer? = ( vector-field3d-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 in 3D space. The arguments are interpreted identically to the corresponding arguments to `vector-field`.

Example:

```
> (plot3d (vector-field3d (λ (x y z) (vector x z y))
                          -2 2 -2 2 -2 2))
```



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 rational? #f) = #f
  x-max : (or/c rational? #f) = #f
  y-min : (or/c rational? #f) = #f
  y-max : (or/c rational? #f) = #f
  z-min : (or/c rational? #f) = #f
  z-max : (or/c rational? #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
```

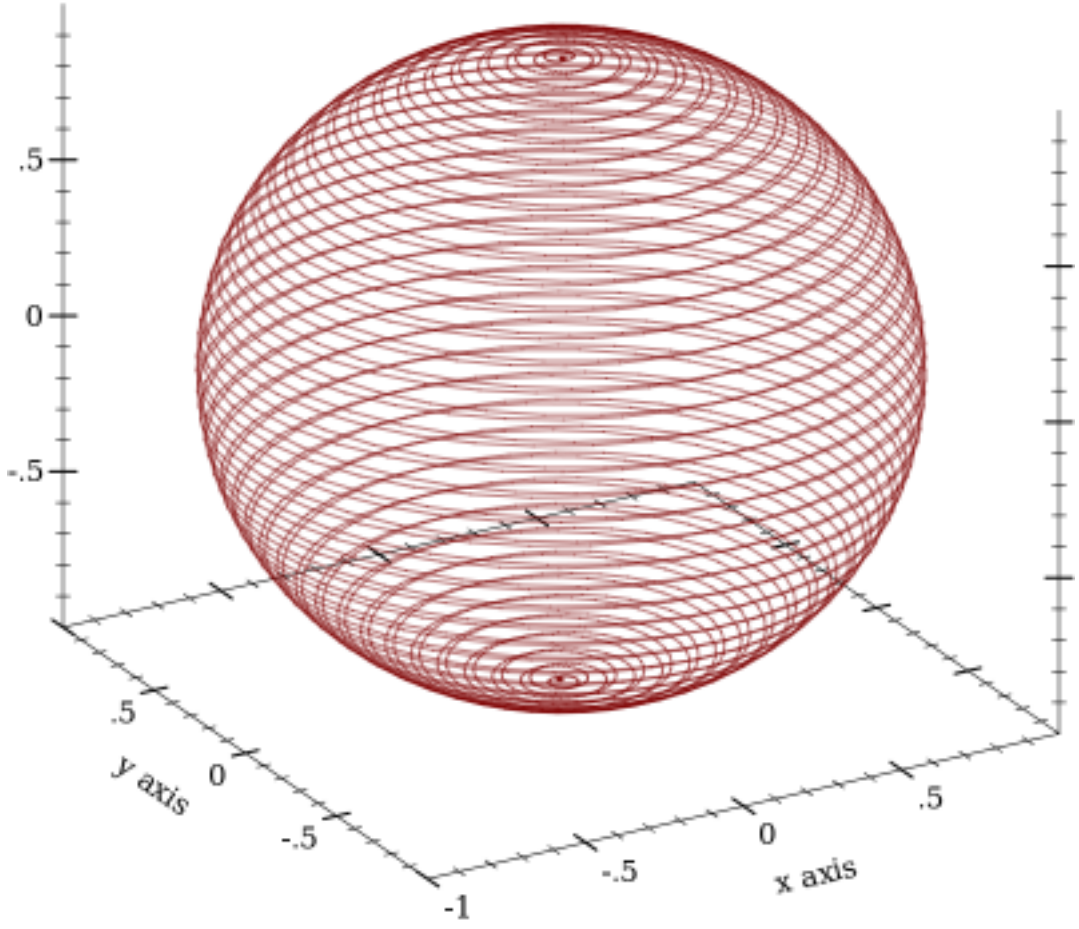
Returns a renderer that draws connected lines. The parametric3d function is defined in terms of this one.

```
(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 : rational?
  t-max : rational?
```

```
x-min : (or/c rational? #f) = #f
x-max : (or/c rational? #f) = #f
y-min : (or/c rational? #f) = #f
y-max : (or/c rational? #f) = #f
z-min : (or/c rational? #f) = #f
z-max : (or/c rational? #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,

```
> (require (only-in plot/utils 3d-polar->3d-cartesian))

> (plot3d (parametric3d (λ (t) (3d-polar->3d-
cartesian (* t 80) t 1))
                        (- pi) pi #:samples 3000 #:alpha 0.5)
          #:altitude 25)
```

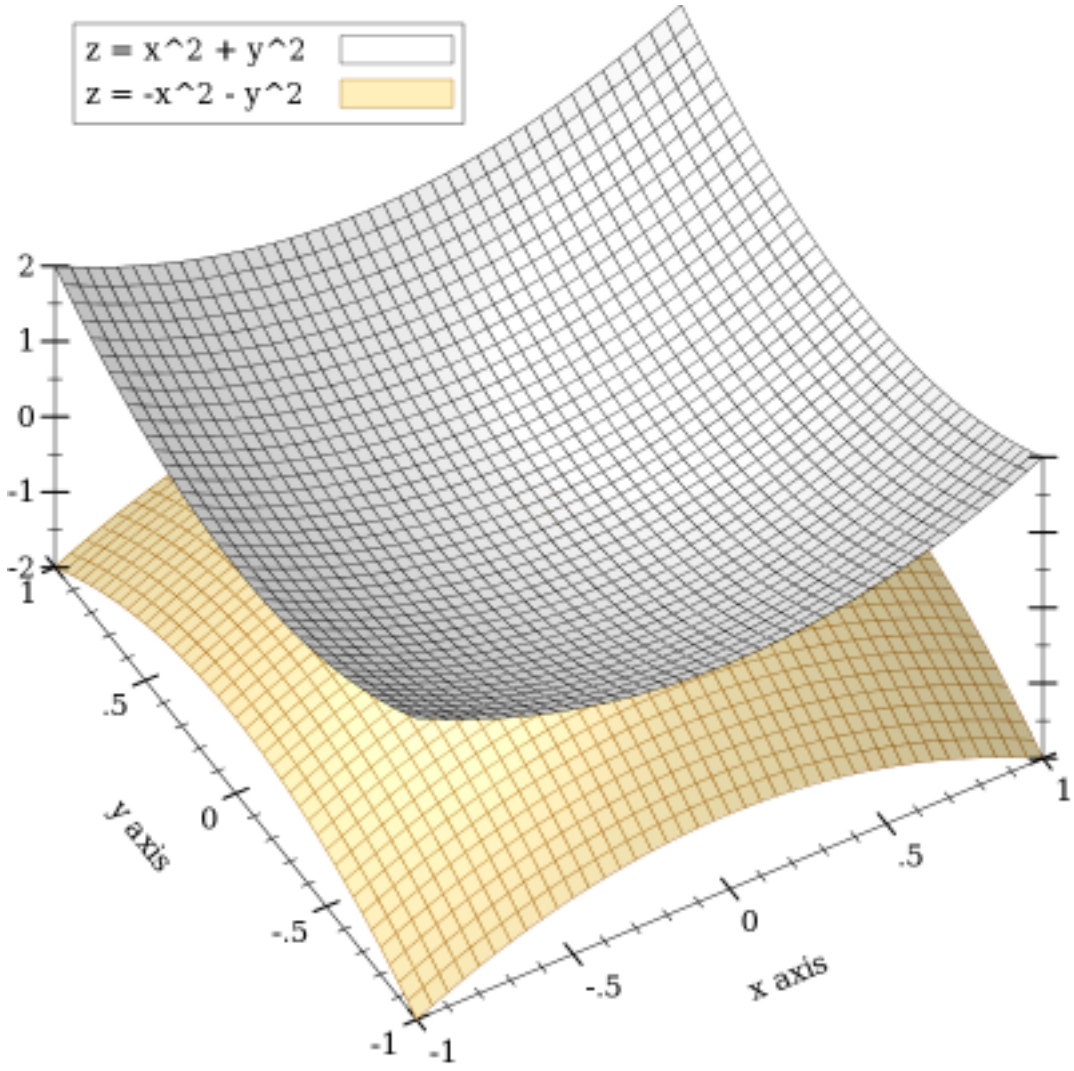


5.4 3D Surface Renderers

```
(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 rational? #f) = #f
  x-max : (or/c rational? #f) = #f
  y-min : (or/c rational? #f) = #f
  y-max : (or/c rational? #f) = #f
  z-min : (or/c rational? #f) = #f
  z-max : (or/c rational? #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
.5
0
-.5
-1
y axis
-1
-.5
0
.5
1
x axis

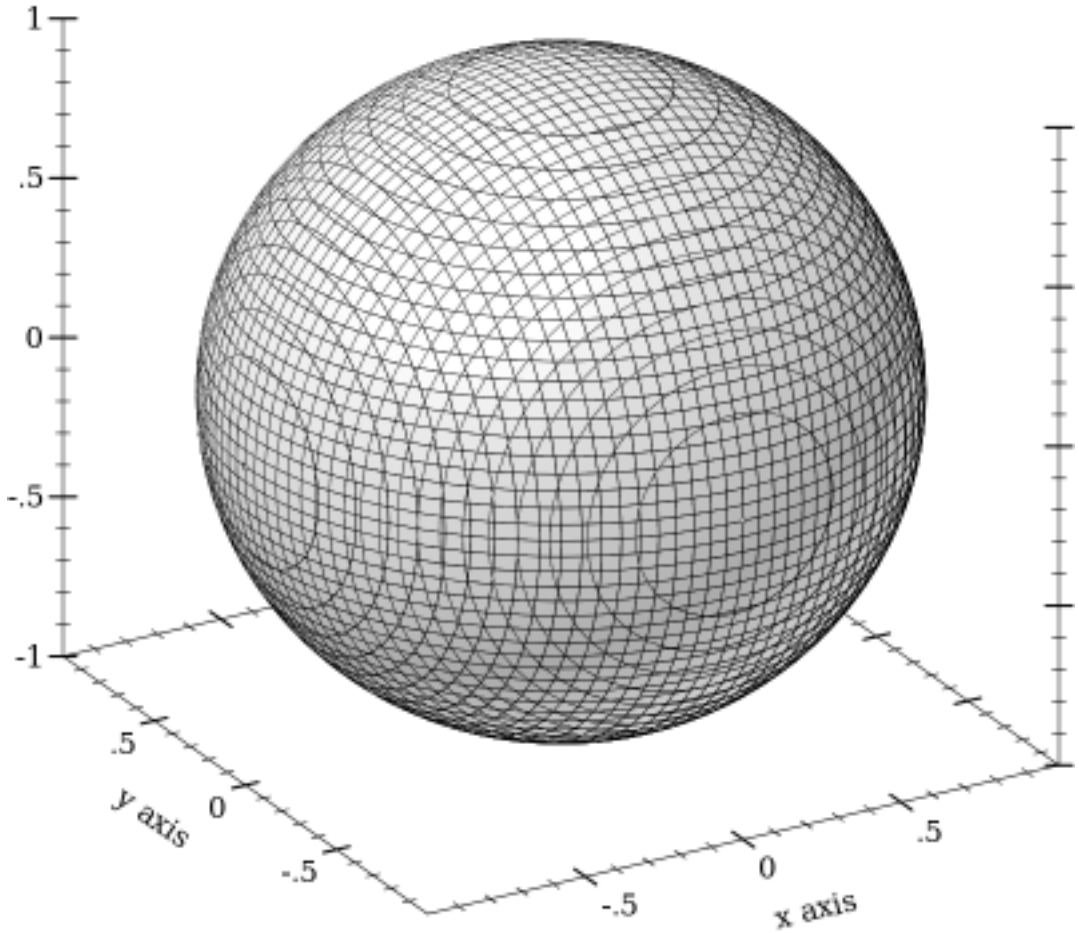
```
(polar3d  f
          [#:x-min x-min
           #:x-max x-max
           #:y-min y-min
           #:y-max y-max
           #:z-min z-min
           #:z-max z-max
           #:samples samples
           #:color color
           #:style style
           #:line-color line-color
           #:line-width line-width
           #:line-style line-style
           #:alpha alpha
           #:label label])          → renderer3d?
  f : (real? real? . -> . real?)
  x-min : (or/c rational? #f) = #f
  x-max : (or/c rational? #f) = #f
  y-min : (or/c rational? #f) = #f
  y-max : (or/c rational? #f) = #f
  z-min : (or/c rational? #f) = #f
  z-max : (or/c rational? #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 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)`. These intervals may become optional arguments to `polar3d` in the future.

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

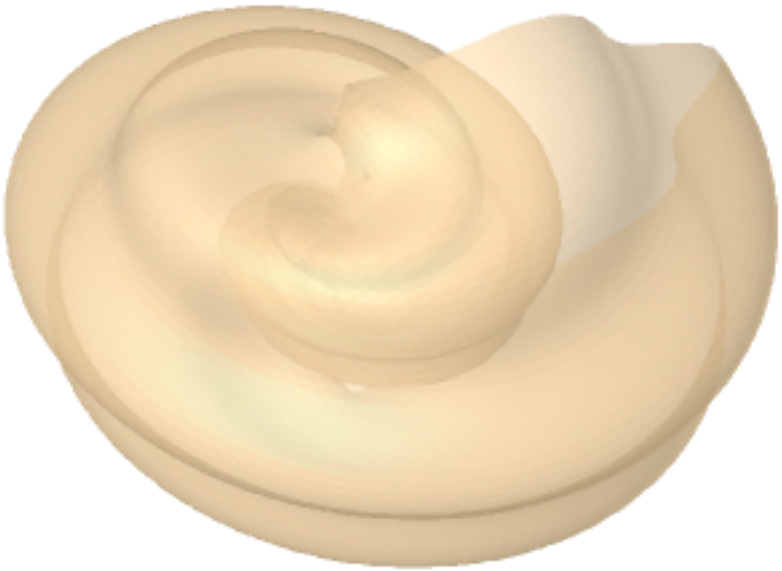
```
> (plot3d (polar3d (λ (θ ρ) 1)) #:altitude 25)
```



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

```
> (parameterize ([plot-decorations?  #f]
                 [plot3d-samples     75])
    (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))))
```



5.5 3D Contour (Isoline) Renderers

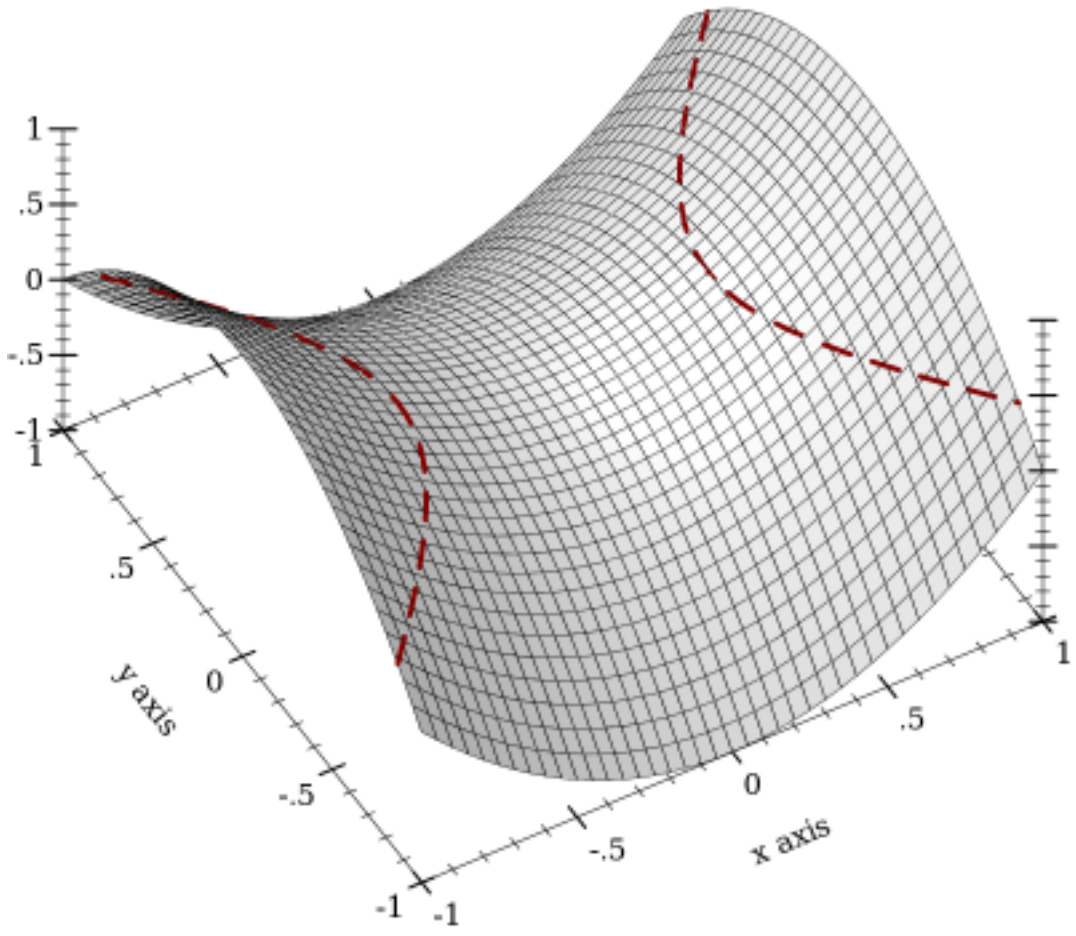
```
(isoline3d  f
            z
           [x-min
            x-max
            y-min
            y-max
            #:z-min z-min
            #:z-max z-max
            #:samples samples
            #:color color
            #:width width
            #:style style
            #:alpha alpha
            #:label label])    → renderer3d?
  f : (real? real? . -> . real?)
  z : real?
  x-min : (or/c rational? #f) = #f
  x-max : (or/c rational? #f) = #f
  y-min : (or/c rational? #f) = #f
  y-max : (or/c rational? #f) = #f
  z-min : (or/c rational? #f) = #f
  z-max : (or/c rational? #f) = #f
  samples : (and/c exact-integer? (>=/c 2)) = (plot3d-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 single contour line on the surface of a function.

The appearance keyword arguments are interpreted identically to the appearance keyword arguments to `isoline`.

This function is not terribly useful by itself, but can be when combined with others:

```
> (define (saddle x y) (- (sqr x) (sqr y)))

> (plot3d (list (surface3d saddle -1 1 -1 1)
                (isoline3d saddle 1/4 #:width 2 #:style 'long-
dash)))
```


1
.5
0
-.5
-1
1
.5
0
-.5
-1
y axis
-1
-.5
0
.5
1
x axis

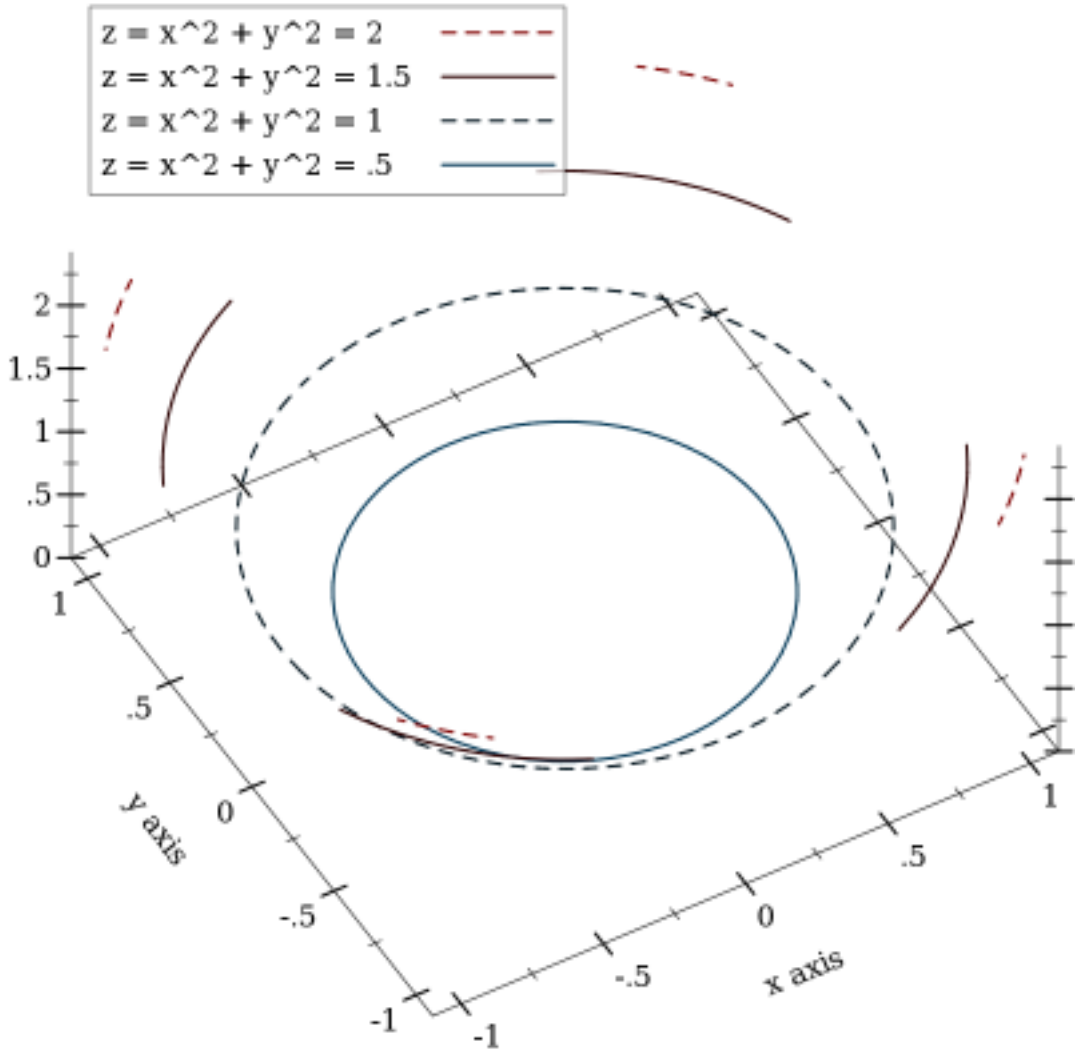
```
(contours3d  f
             [x-min
              x-max
              y-min
              y-max
              #:z-min z-min
              #:z-max z-max
              #:samples samples
              #:levels levels
              #:colors colors
              #:widths widths
              #:styles styles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


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
1.5
1
.5
0
1
.5
0
-.5
-1
y axis
-1
-.5
0
.5
1
x axis

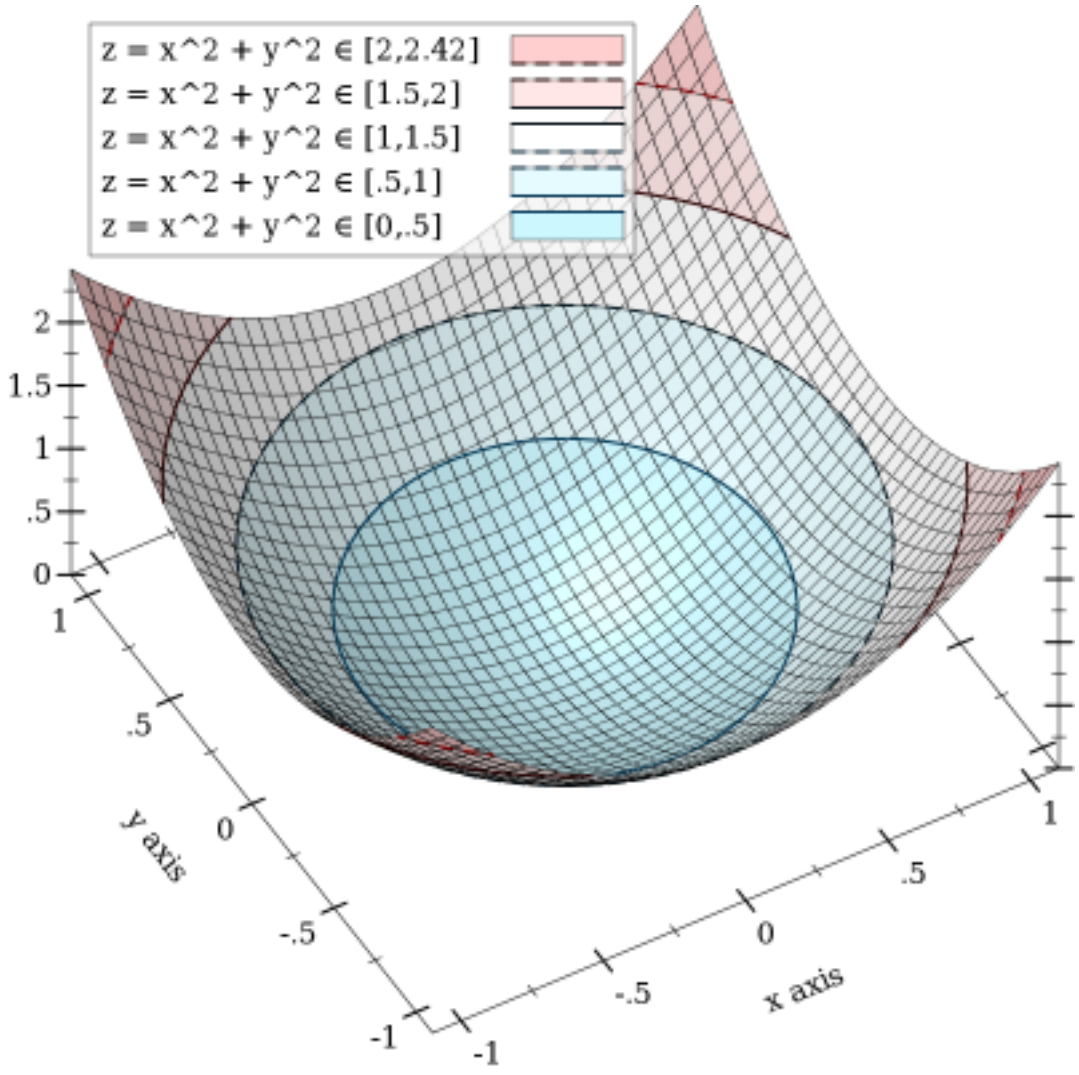
```
(contour-intervals3d  f
                      [x-min
                       x-max
                       y-min
                       y-max
                       #:z-min z-min
                       #:z-max z-max
                       #:samples samples
                       #:levels levels
  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

```
alphas : (alphas/c (listof ivl?)) = (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"))
```



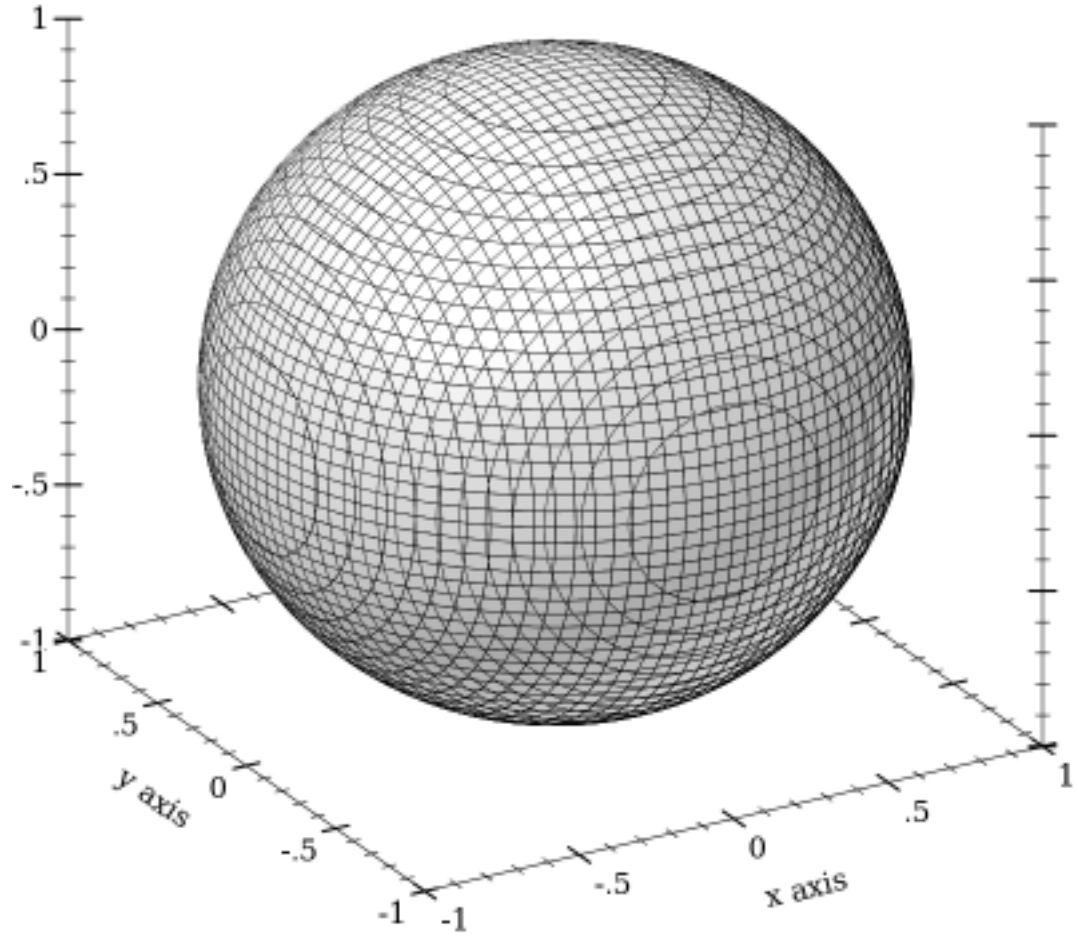
5.6 3D Isosurface Renderers

```
(isosurface3d  f
               d
              [x-min
               x-max
               y-min
               y-max
               z-min
               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? . -> . real?)
  d : rational?
  x-min : (or/c rational? #f) = #f
  x-max : (or/c rational? #f) = #f
  y-min : (or/c rational? #f) = #f
  y-max : (or/c rational? #f) = #f
  z-min : (or/c rational? #f) = #f
  z-max : (or/c rational? #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 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)
```

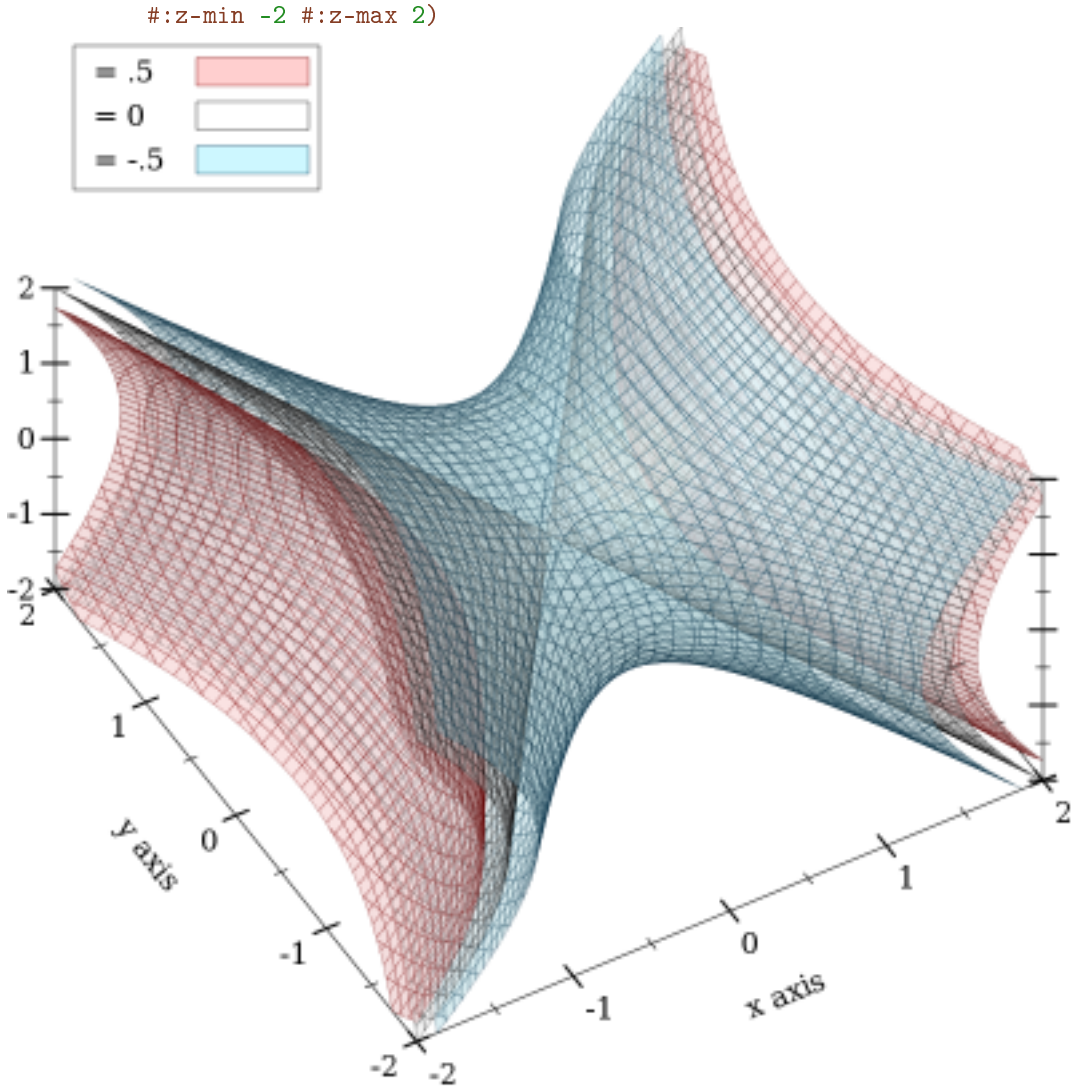

1
.5
0
-.5
-1
1
.5
0
-.5
-1
y axis
-1
-.5
0
.5
1
x axis

```
(isosurfaces3d  f
                [x-min
                 x-max
                 y-min
                 y-max
                 z-min
                 z-max
                 #:d-min d-min
                 #:d-max d-max
                 #:samples samples
                 #:levels levels
                 #:colors colors
                 #:styles styles
                 #: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 rational? #f) = #f
  x-max : (or/c rational? #f) = #f
  y-min : (or/c rational? #f) = #f
  y-max : (or/c rational? #f) = #f
  z-min : (or/c rational? #f) = #f
  z-max : (or/c rational? #f) = #f
  d-min : (or/c rational? #f) = #f
  d-max : (or/c rational? #f) = #f
  samples : (and/c exact-integer? (>=/c 2)) = (plot3d-samples)
  levels : (or/c 'auto exact-positive-integer? (listof real?))
         = (isosurface-levels)
  colors : (plot-colors/c (listof real?)) = (isosurface-colors)
  styles : (plot-brush-styles/c (listof real?))
         = (isosurface-styles)
  line-colors : (plot-colors/c (listof real?))
              = (isosurface-line-colors)
  line-widths : (pen-widths/c (listof real?))
              = (isosurface-line-widths)
  line-styles : (plot-pen-styles/c (listof real?))
              = (isosurface-line-styles)
  alphas : (alphas/c (listof real?)) = (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:

```
> (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)
```



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

5.7 3D Rectangle Renderers

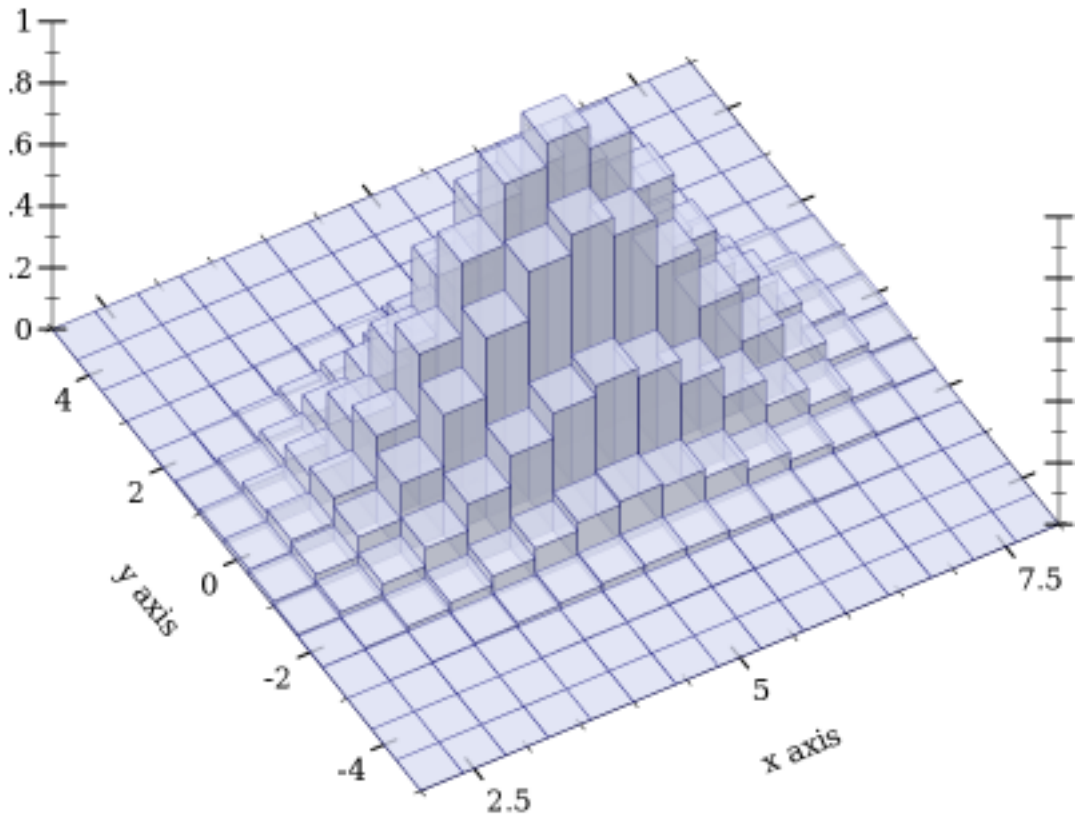
```
(rectangles3d  rects
               [#:x-min x-min
                #:x-max x-max
                #:y-min y-min
                #:y-max y-max
                #:z-min z-min
                #:z-max z-max
                #:color color
                #:style style
                #:line-color line-color
                #:line-width line-width
                #:line-style line-style
                #:alpha alpha
                #:label label])          → renderer3d?
  rects : (listof (vector/c ivl? ivl? ivl?))
  x-min : (or/c rational? #f) = #f
  x-max : (or/c rational? #f) = #f
  y-min : (or/c rational? #f) = #f
  y-max : (or/c rational? #f) = #f
  z-min : (or/c rational? #f) = #f
  z-max : (or/c rational? #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,

```
> (require (only-in plot/utils bounds->intervals linear-seq))

> (define (norm2 x y) (exp (* -1/2 (+ (sqr (- x 5)) (sqr y)))))

> (define x-ivls (bounds->intervals (linear-seq 2 8 16)))

> (define y-ivls (bounds->intervals (linear-seq -5 5 16)))

> (define x-mids (linear-seq 2 8 15 #:start? #f #:end? #f))
```

```
> (define y-mids (linear-seq -5 5 15 #: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"))
```



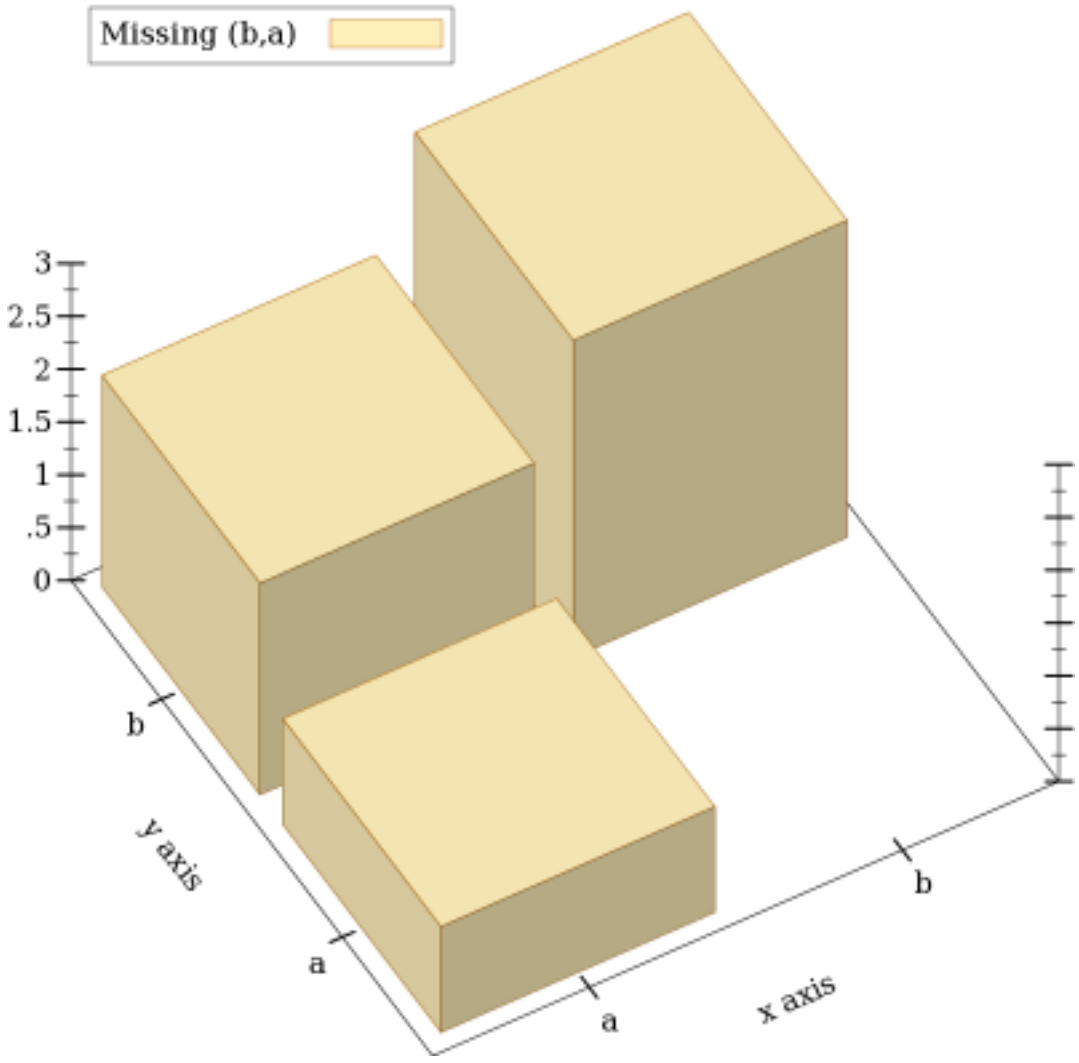
```
(discrete-histogram3d  cat-vals
                       [#:x-min x-min
                        #:x-max x-max
                        #:y-min y-min
                        #:y-max y-max
                        #:z-min z-min
                        #:z-max z-max
                        #:gap gap
                        #:color color
                        #:style style
                        #:line-color line-color
                        #:line-width line-width
                        #:line-style line-style
                        #:alpha alpha
                        #:label label
                        #:add-x-ticks? add-x-ticks?
                        #:add-y-ticks? add-y-ticks?
                        #:x-far-ticks? x-far-ticks?
                        #:y-far-ticks? y-far-ticks?])
 → renderer3d?
  cat-vals : (listof (vector/c any/c any/c (or/c real? ivl? #f)))
  x-min : (or/c rational? #f) = 0
  x-max : (or/c rational? #f) = #f
  y-min : (or/c rational? #f) = 0
  y-max : (or/c rational? #f) = #f
  z-min : (or/c rational? #f) = 0
  z-max : (or/c rational? #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
  add-x-ticks? : boolean? = #t
  add-y-ticks? : boolean? = #t
  x-far-ticks? : boolean? = #f
  y-far-ticks? : boolean? = #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))
```



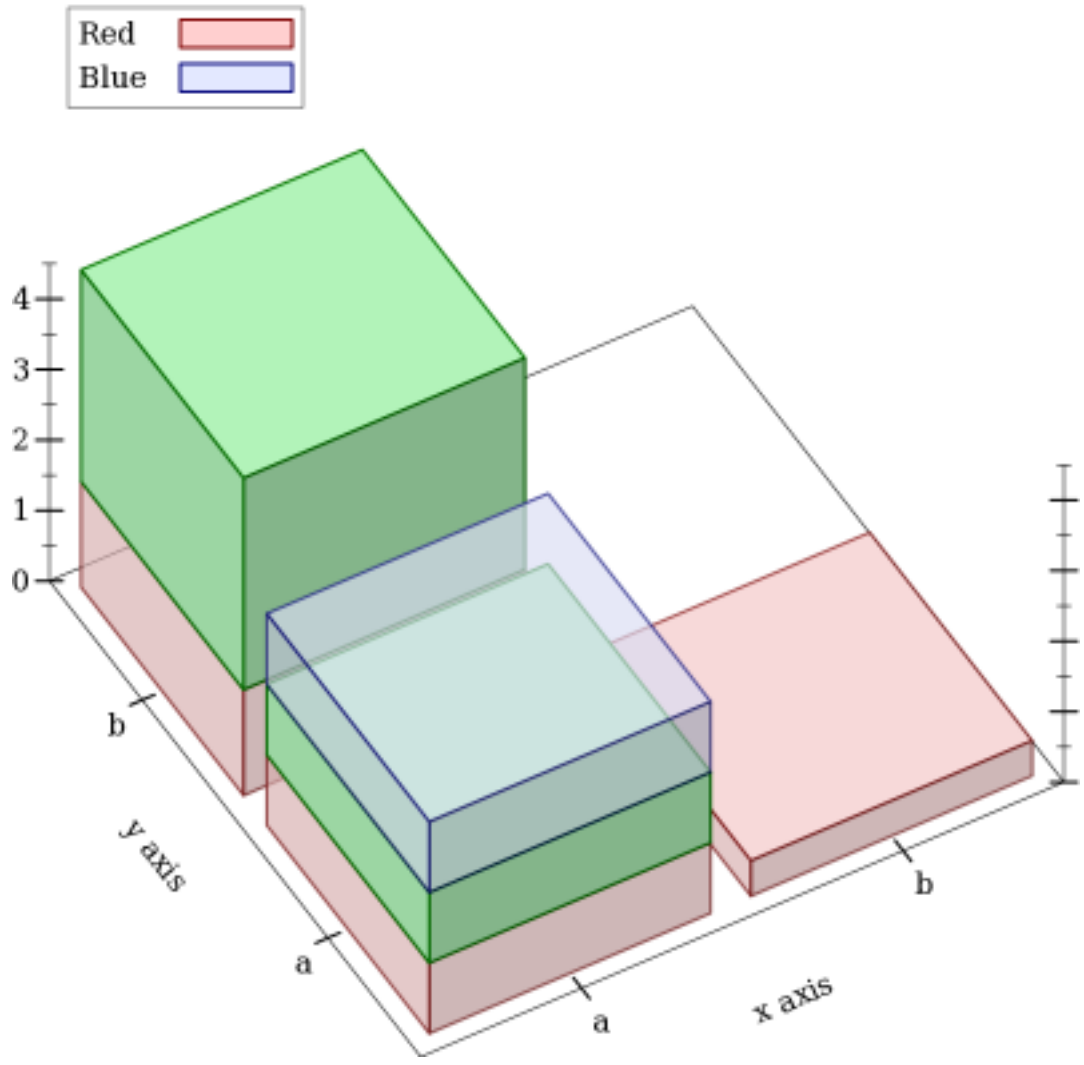
```
(stacked-histogram3d  cat-vals
                      [#:x-min x-min
                       #:x-max x-max
                       #:y-min y-min
                       #:y-max y-max
                       #:z-min z-min
                       #:z-max z-max
                       #:gap gap
                       #:colors colors
                       #:styles styles
                       #:line-colors line-colors
                       #:line-widths line-widths
                       #:line-styles line-styles
                       #:alphas alphas
                       #:labels labels
                       #:add-x-ticks? add-x-ticks?
                       #:add-y-ticks? add-y-ticks?
                       #:x-far-ticks? x-far-ticks?
                       #:y-far-ticks? y-far-ticks?])
 → (listof renderer3d?)
  cat-vals : (listof (vector/c any/c any/c (listof real?)))
  x-min : (or/c rational? #f) = 0
  x-max : (or/c rational? #f) = #f
  y-min : (or/c rational? #f) = 0
  y-max : (or/c rational? #f) = #f
  z-min : (or/c rational? #f) = 0
  z-max : (or/c rational? #f) = #f
  gap : (real-in 0 1) = (discrete-histogram-gap)
  colors : (plot-colors/c nat/c) = (stacked-histogram-colors)
  styles : (plot-brush-styles/c nat/c)
         = (stacked-histogram-styles)
  line-colors : (plot-colors/c nat/c)
              = (stacked-histogram-line-colors)
  line-widths : (pen-widths/c nat/c)
              = (stacked-histogram-line-widths)
  line-styles : (plot-pen-styles/c nat/c)
              = (stacked-histogram-line-styles)
  alphas : (alphas/c nat/c) = (stacked-histogram-alphas)
  labels : (labels/c nat/c) = '(#f)
  add-x-ticks? : boolean? = #t
  add-y-ticks? : boolean? = #t
  x-far-ticks? : boolean? = #f
  y-far-ticks? : boolean? = #f
```

Returns a renderer that draws a stacked histogram. Think of it as a version of discrete-

histogram that allows multiple values to be specified for each pair of categories.

Examples:

```
> (define data '(#(a a (1 1 1)) #(a b (1.5 3)) #(b b ()) #(b a (1/2))))

> (plot3d (stacked-histogram3d data #:labels '("Red" #f "Blue")
                               #:alphas '(2/3 1 2/3)))
```



6 Nonrenderers

The following functions create *nonrenderers*, or plot elements that draw nothing in the plot.

```
(x-ticks ts [#:far? far?]) → nonrenderer?
  ts : (listof tick?)
  far? : boolean? = #f
```

```
(y-ticks ts [#:far? far?]) → nonrenderer?
  ts : (listof tick?)
  far? : boolean? = #f
```

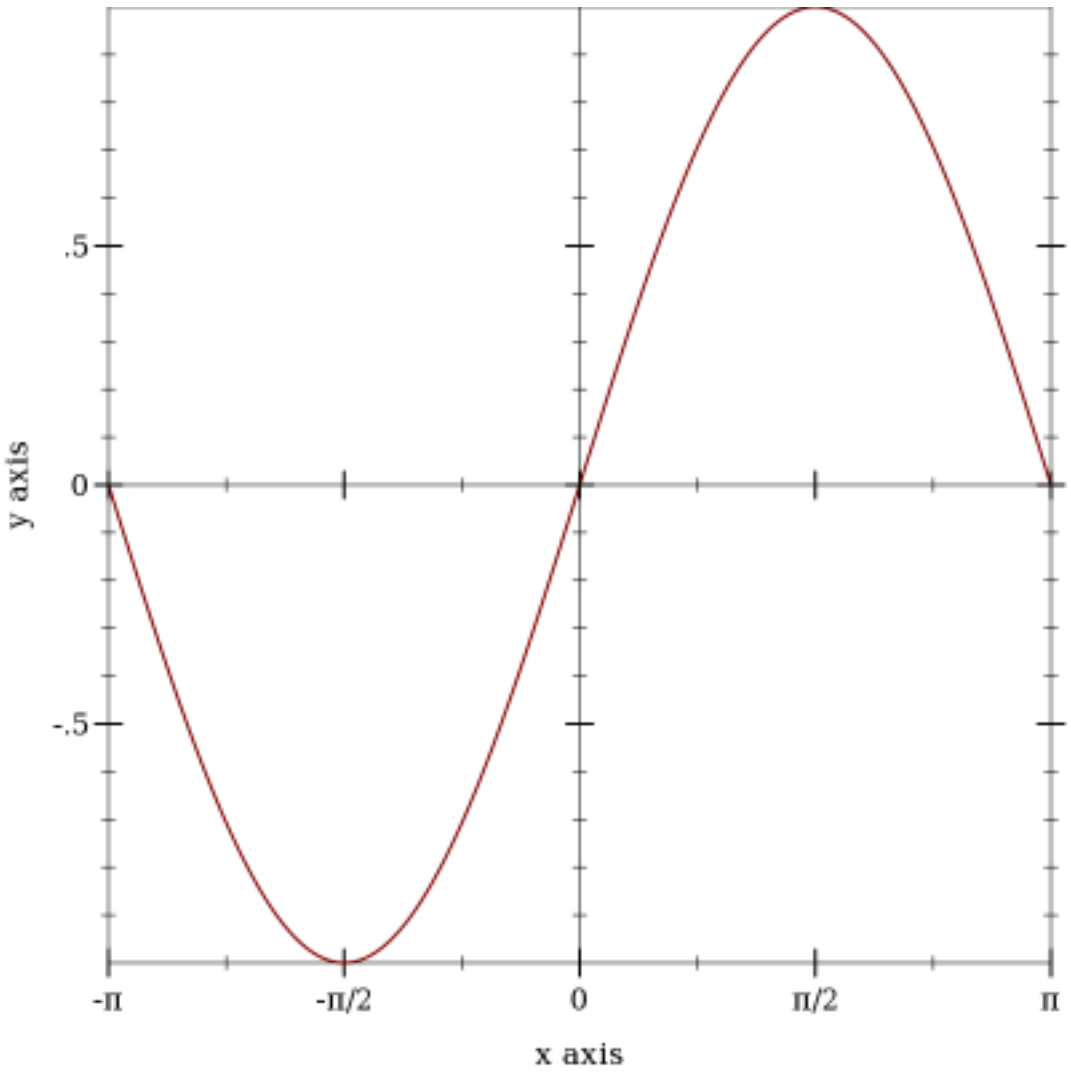
```
(z-ticks ts [#:far? far?]) → nonrenderer?
  ts : (listof tick?)
  far? : boolean? = #f
```

The `x-ticks`, `y-ticks` and `z-ticks` return a nonrenderer that adds custom ticks to a 2D or 3D plot.

Although `ticks-add` allows placing arbitrary major and minor ticks on an axis, it does not allow them to be formatted differently from the other ticks on the same axis. Use one of these functions to get maximum control.

Example:

```
> (parameterize ([plot-x-ticks  no-ticks])
    (plot (list (function sin (- pi) pi)
                (x-ticks (list (tick (- pi) #t "-π")
                               (tick (* -3/4 pi) #f "")
                               (tick (* -1/2 pi) #t "-π/2")
                               (tick (* -1/4 pi) #f "")
                               (tick 0 #t "0")
                               (tick (* 1/4 pi) #f "")
                               (tick (* 1/2 pi) #t "π/2")
                               (tick (* 3/4 pi) #f "")
                               (tick pi #t "π")))
                (axes))))
```



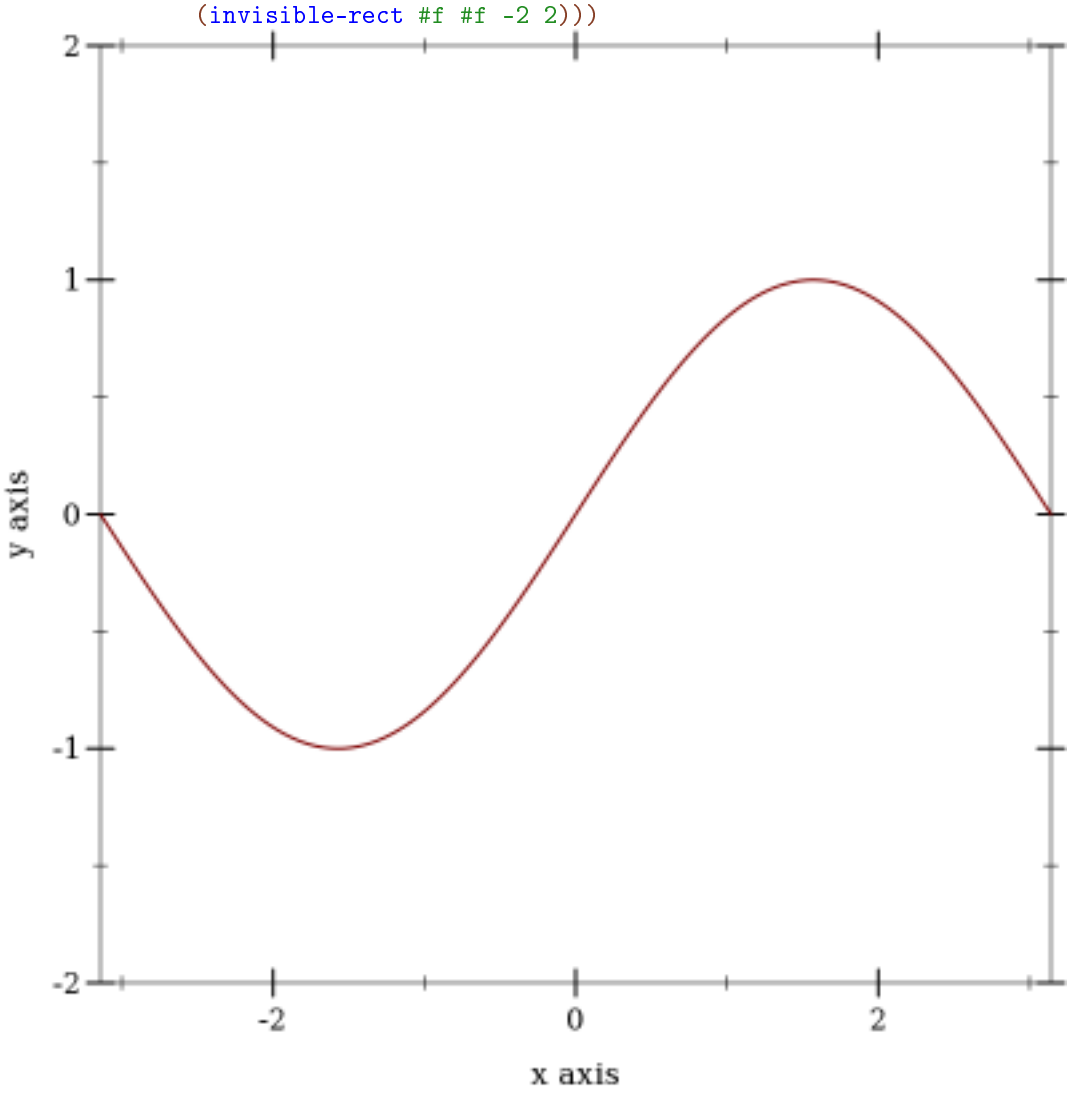
When considering using one of these functions, remember that minor tick labels are never drawn, and that including a `z-ticks` nonrenderer will not add extra contour lines to contour plots.

```
(invisible-rect x-min x-max y-min y-max) → nonrenderer?
  x-min : (or/c rational? #f)
  x-max : (or/c rational? #f)
  y-min : (or/c rational? #f)
  y-max : (or/c rational? #f)
```

Returns a nonrenderer that simply takes up space in the plot. Use this to cause the plot area to include a minimal rectangle.

Example:

```
> (plot (list (function sin (- pi) pi)
              (invisible-rect #f #f -2 2)))
```




```
(invisible-rect3d x-min
                  x-max
                  y-min
                  y-max
                  z-min
                  z-max) → nonrenderer?
  x-min : (or/c rational? #f)
  x-max : (or/c rational? #f)
```

```
y-min : (or/c rational? #f)
y-max : (or/c rational? #f)
z-min : (or/c rational? #f)
z-max : (or/c rational? #f)
```

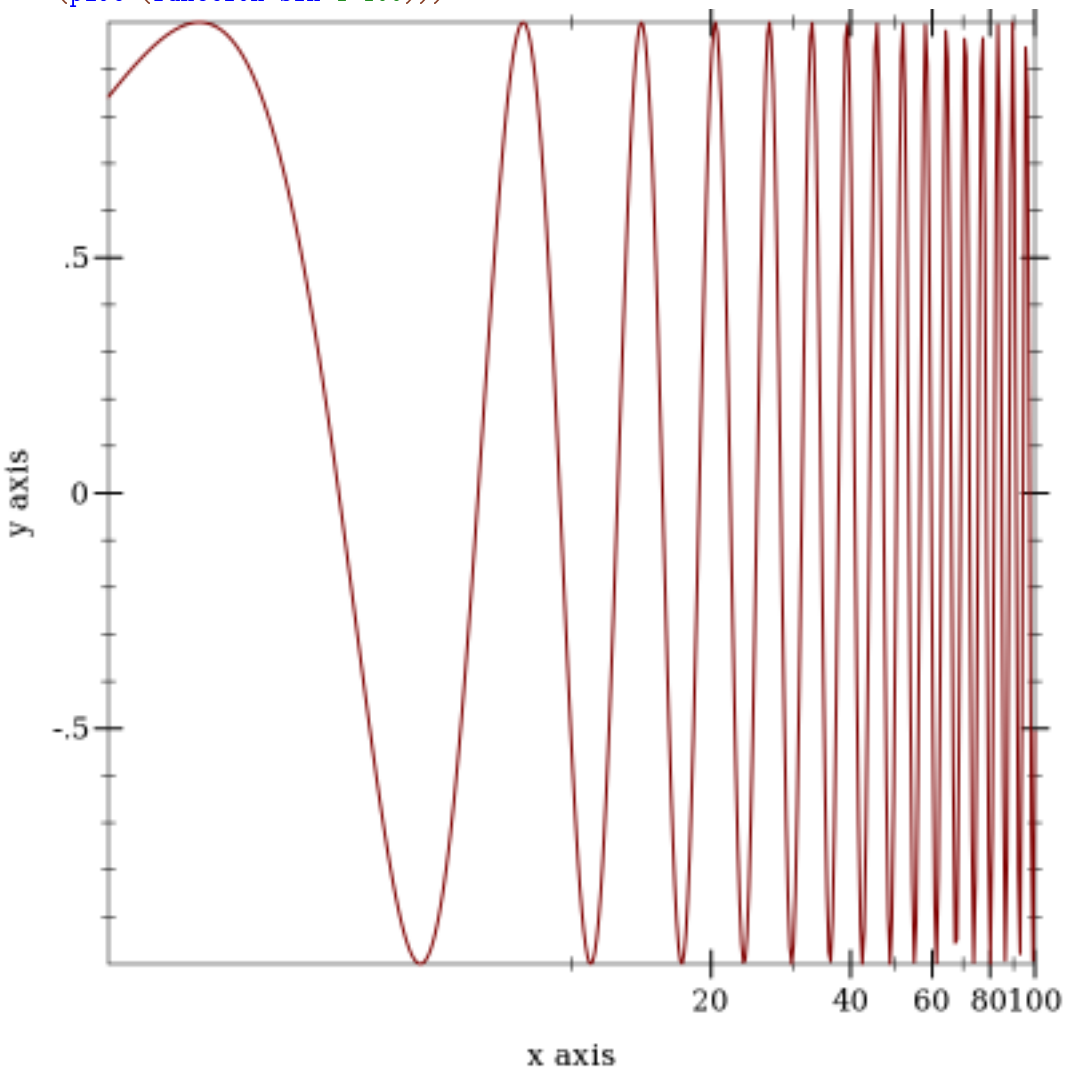
Returns a nonrenderer that simply takes up space in the plot. Use this to cause the plot area to include a minimal rectangle. See `invisible-rect` for a 2D example.

7 Axis Transforms and Ticks

7.1 Axis Transforms

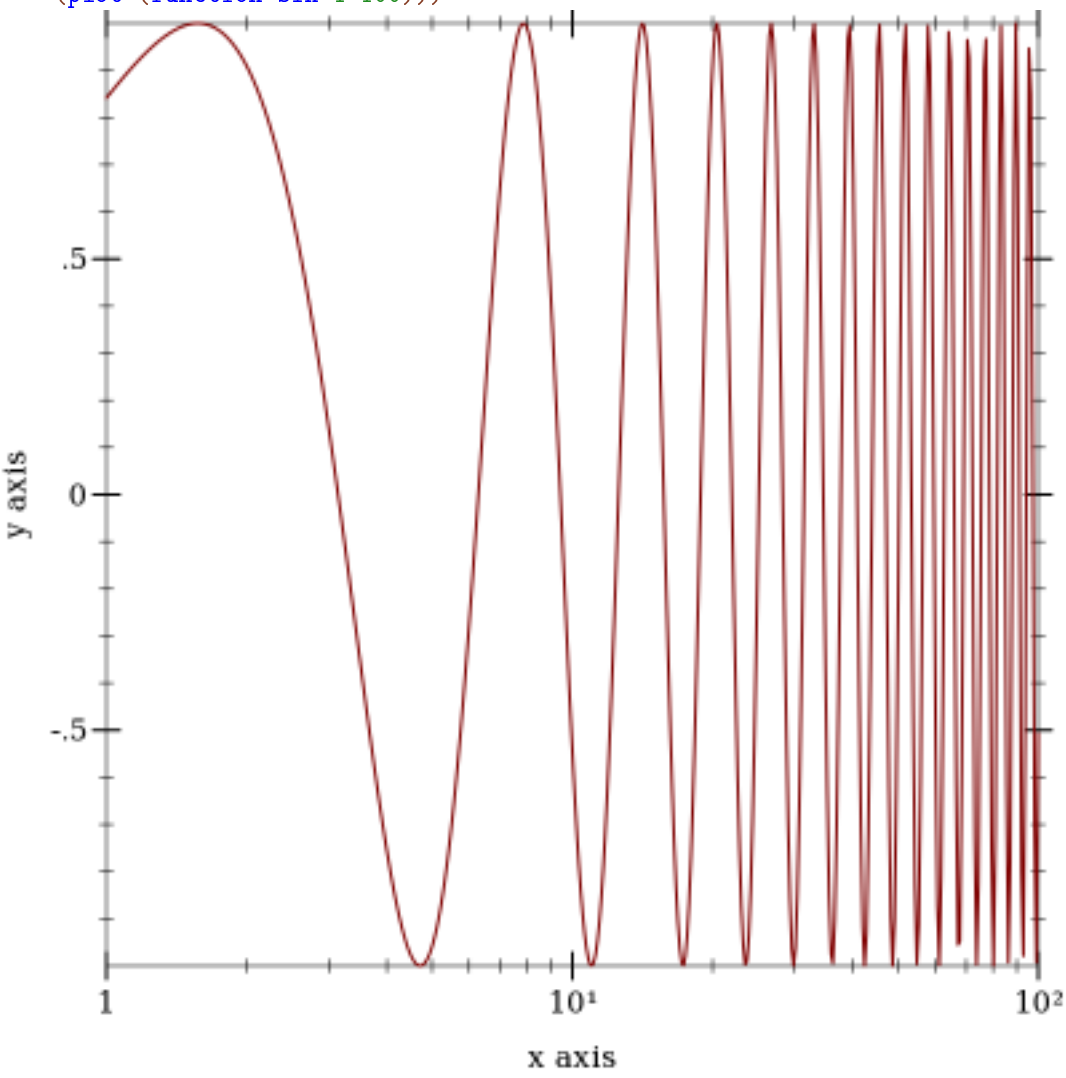
The *x*, *y* and *z* axes for any plot can be independently transformed by parameterizing the plot on different `plot-x-transform`, `plot-y-transform` and `plot-z-transform` values. For example, to plot the *x* axis with a log transform:

```
> (parameterize ([plot-x-transform  log-transform])
    (plot (function sin 1 100)))
```



Most log-transformed plots use different ticks than the default, uniformly spaced ticks, however. To put log ticks on the *x* axis, set the plot-x-ticks parameter:

```
> (parameterize ([plot-x-transform  log-transform]
                 [plot-x-ticks      (log-ticks)])
    (plot (function sin 1 100)))
```

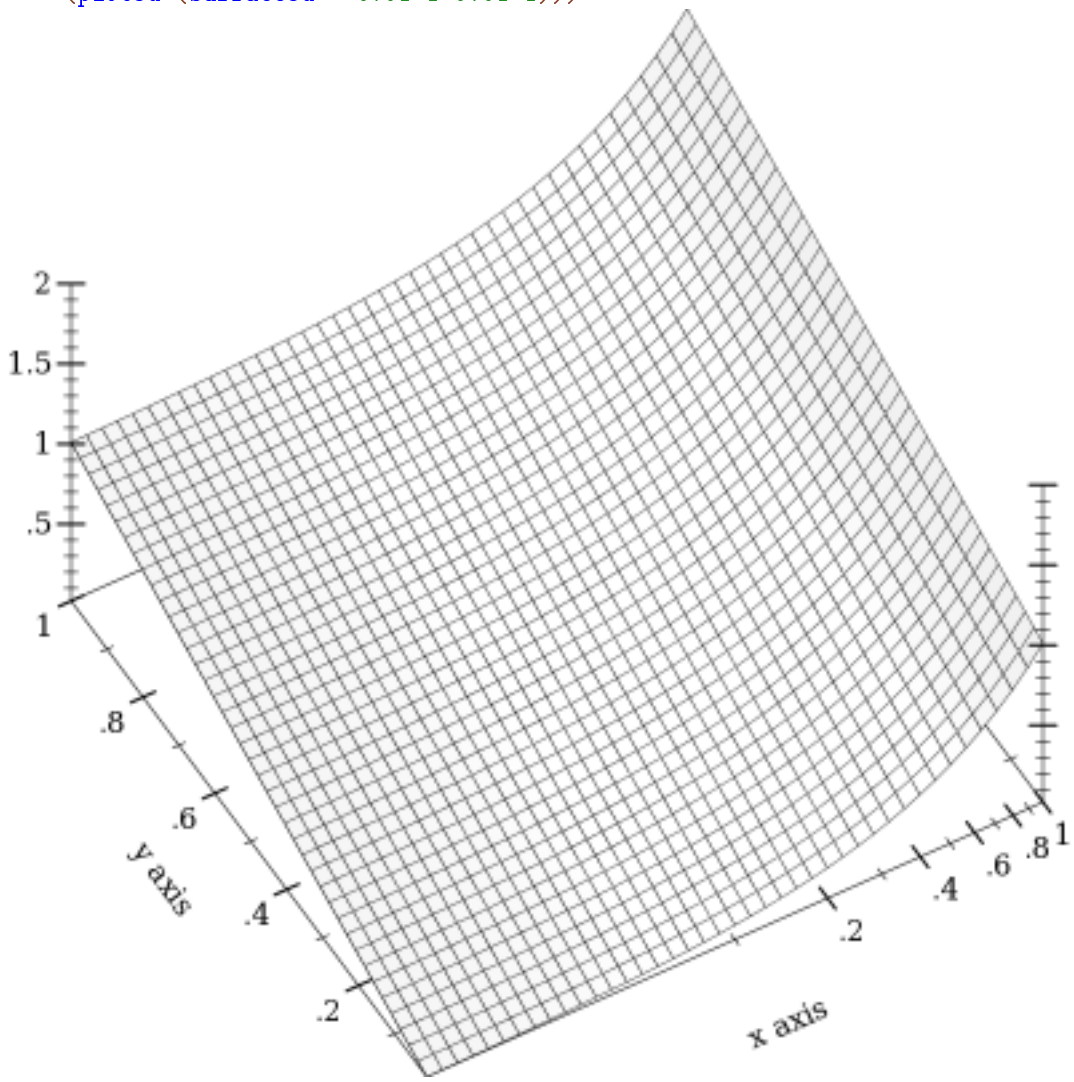


See §7.2 “Axis Ticks” for more details on parameterizing a plot’s axis ticks.

Renderers cooperate with the current transforms by sampling nonlinearly. For example,

To sample nonlinearly, the *inverse* of a transform is applied to linearly sampled points. See make-axis-transform and nonlinear-seq.

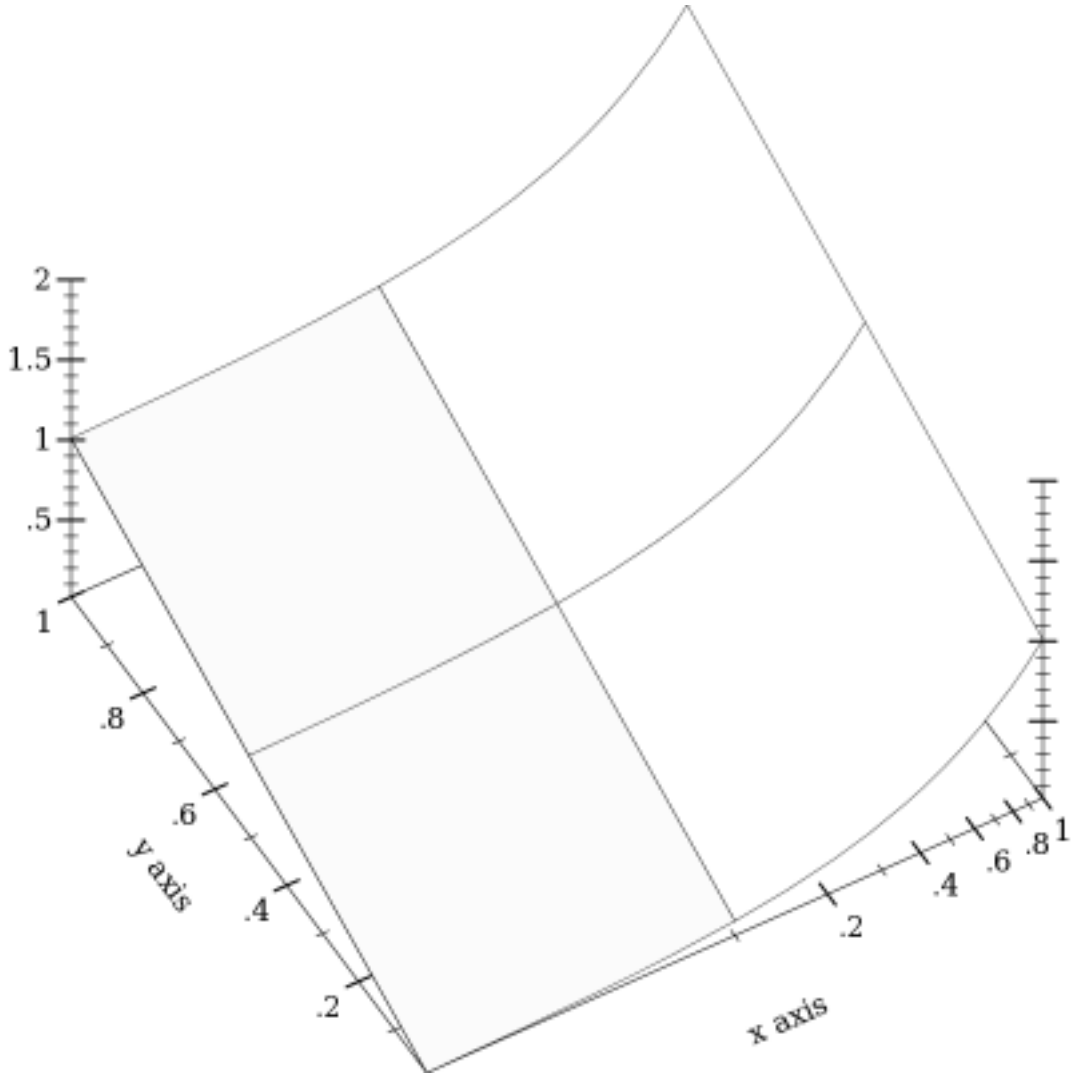
```
> (parameterize ([plot-x-transform  log-transform])
    (plot3d (surface3d + 0.01 1 0.01 1)))
```



Notice that the surface is sampled uniformly in appearance even though the *x*-axis ticks are not spaced uniformly.

Transforms are applied to the primitive shapes that comprise a plot:

```
> (parameterize ([plot-x-transform  log-transform])
    (plot3d (surface3d + 0.01 1 0.01 1 #:samples 3)))
```



Here, the renderer returned by `surface3d` does not have to bend the polygons it draws; `plot3d` does this automatically (by recursive subdivision).

```
(plot-x-transform) → axis-transform/c
(plot-x-transform transform) → void?
  transform : axis-transform/c

=  id-transform
```

```
(plot-y-transform) → axis-transform/c
(plot-y-transform transform) → void?
  transform : axis-transform/c

=  id-transform
```

```
(plot-z-transform) → axis-transform/c
(plot-z-transform transform) → void?
  transform : axis-transform/c

=  id-transform
```

Independent, per-axis, monotone, nonlinear transforms. PLoT comes with some typical (and some atypical) axis transforms, documented immediately below.

```
id-transform : axis-transform/c
```

The identity axis transform, the default transform for all axes.

```
log-transform : axis-transform/c
```

A log transform. Use this to generate plots with log-scale axes. Any such axis must have positive bounds.

The beginning of the §7 "Axis Transforms and Ticks" section has a working example. An example of exceeding the bounds is

```
> (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
```

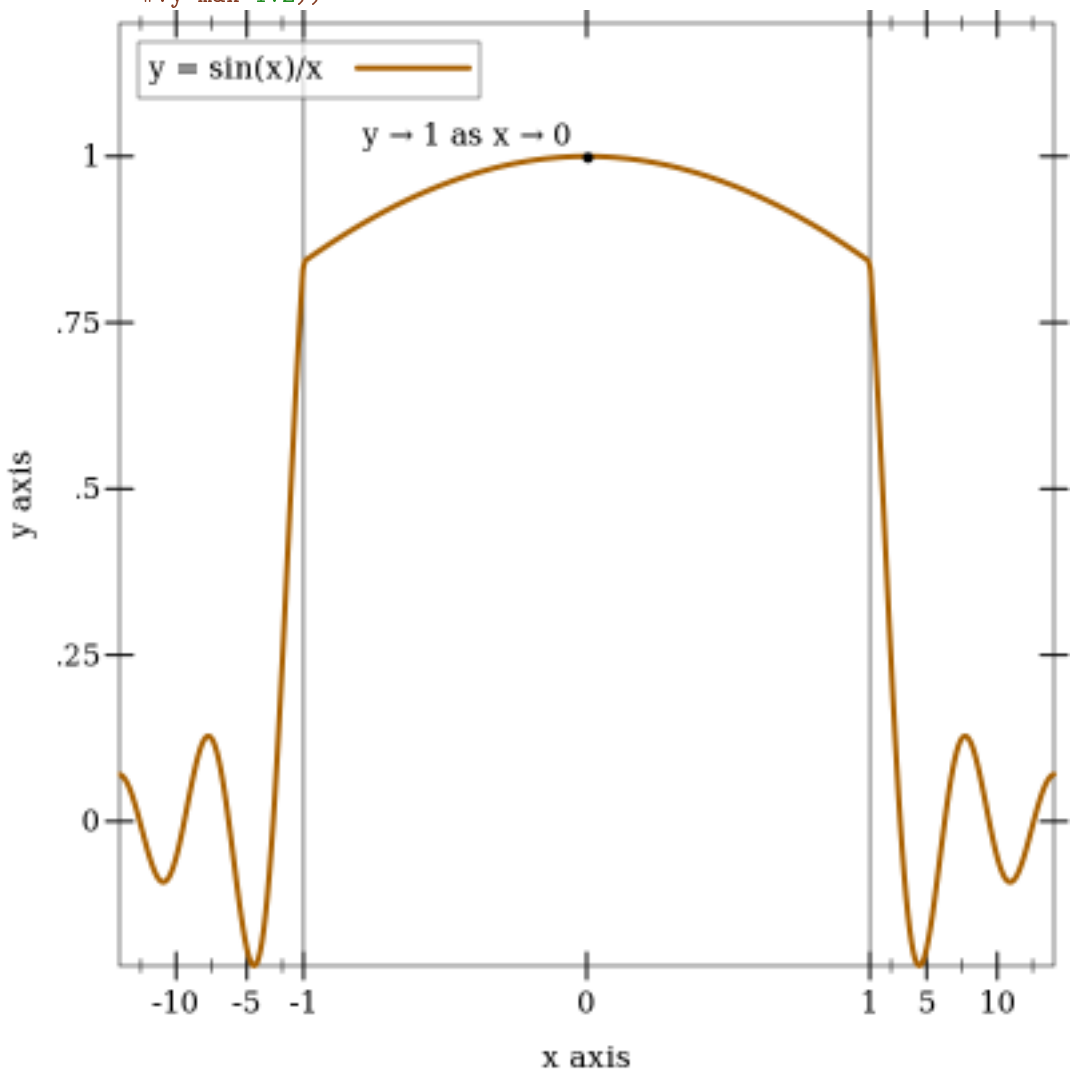
See `axis-transform-bound` and `axis-transform-append` for ways to get around an axis transform's bounds limitations.

```
(stretch-transform a b scale) → axis-transform/c
  a : real?
  b : real?
  scale : (>/c 0)
```

Returns an axis transform that stretches a finite interval.

The following example uses a `stretch-transform` to draw attention to the interval [-1,1] in an illustration of the limit of *sin(x)/x* as *x* approaches zero (a critical part of proving the derivative of *sin(x)*):

```
> (parameterize ([plot-x-transform  (stretch-transform -1 1 20)]
                 [plot-x-ticks  (ticks-add (plot-x-
ticks) '(-1 1))])
    (plot (list (y-axis -1 #:ticks? #f) (y-axis 1 #:ticks? #f)
                (function (λ (x) (/ (sin x) x)) -14 14
                          #:width 2 #:color 4 #:label "y =
sin(x)/x")
                (point-label (vector 0 1) "y → 1 as x → 0"
                             #:anchor 'bottom-right))
          #:y-max 1.2))
```

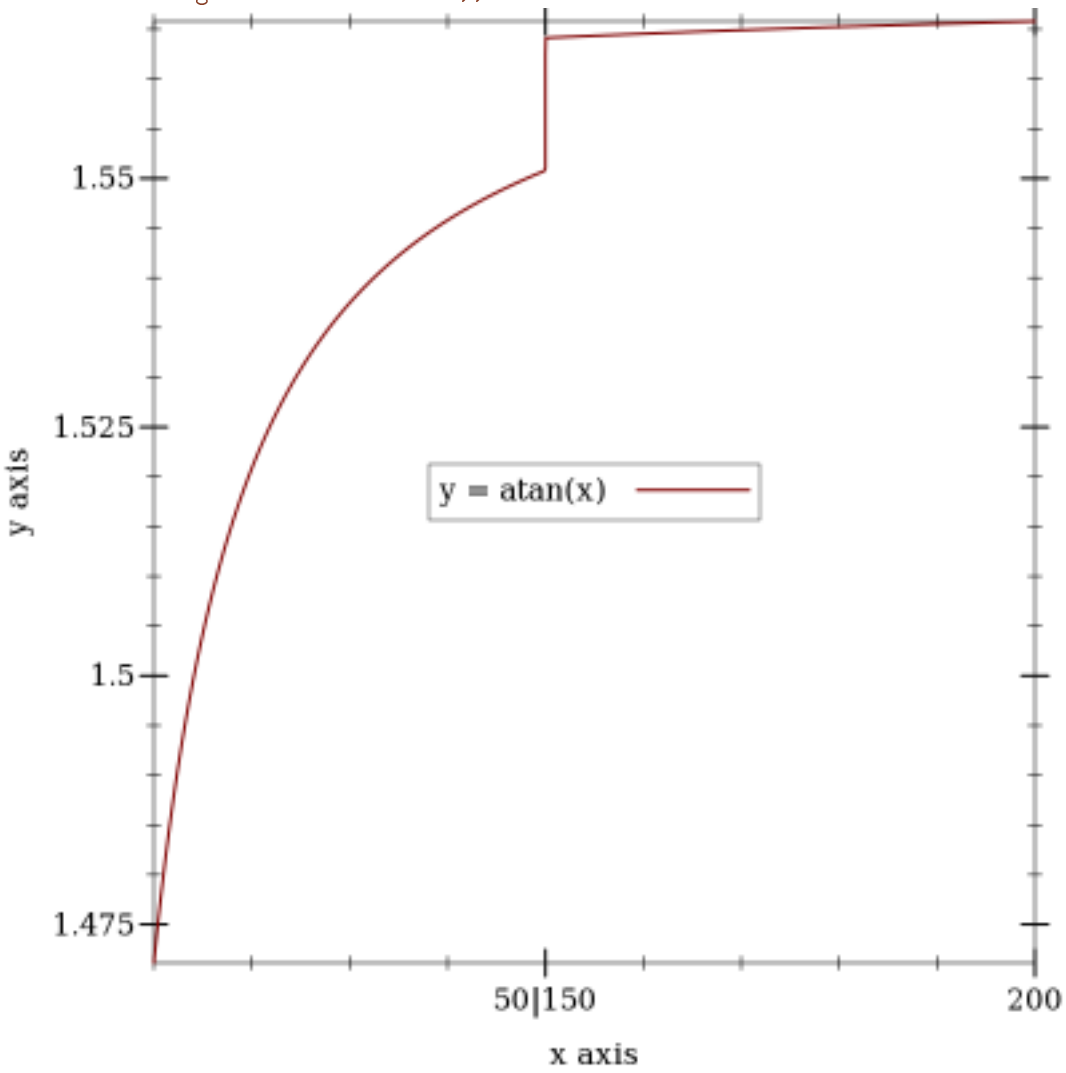



```
(collapse-transform a b) → axis-transform/c
```

```
a : real?
b : real?
```

Returns an axis transform that collapses a finite interval to its midpoint. For example, to remove part of the long, boring asymptotic approach of *atan(x)* toward $\pi/2$:

```
> (parameterize ([plot-x-transform  (collapse-transform 50 150)])
    (plot (function atan 10 200 #:label "y = atan(x)")
          #:legend-anchor 'center))
```



In this case, there were already ticks at the collapsed interval's endpoints. If there had not

been, it would have been necessary to use `ticks-add` to let viewers know precisely the interval that was collapsed. (See `stretch-transform` for an example.)

```
cbrt-transform : axis-transform/c
```

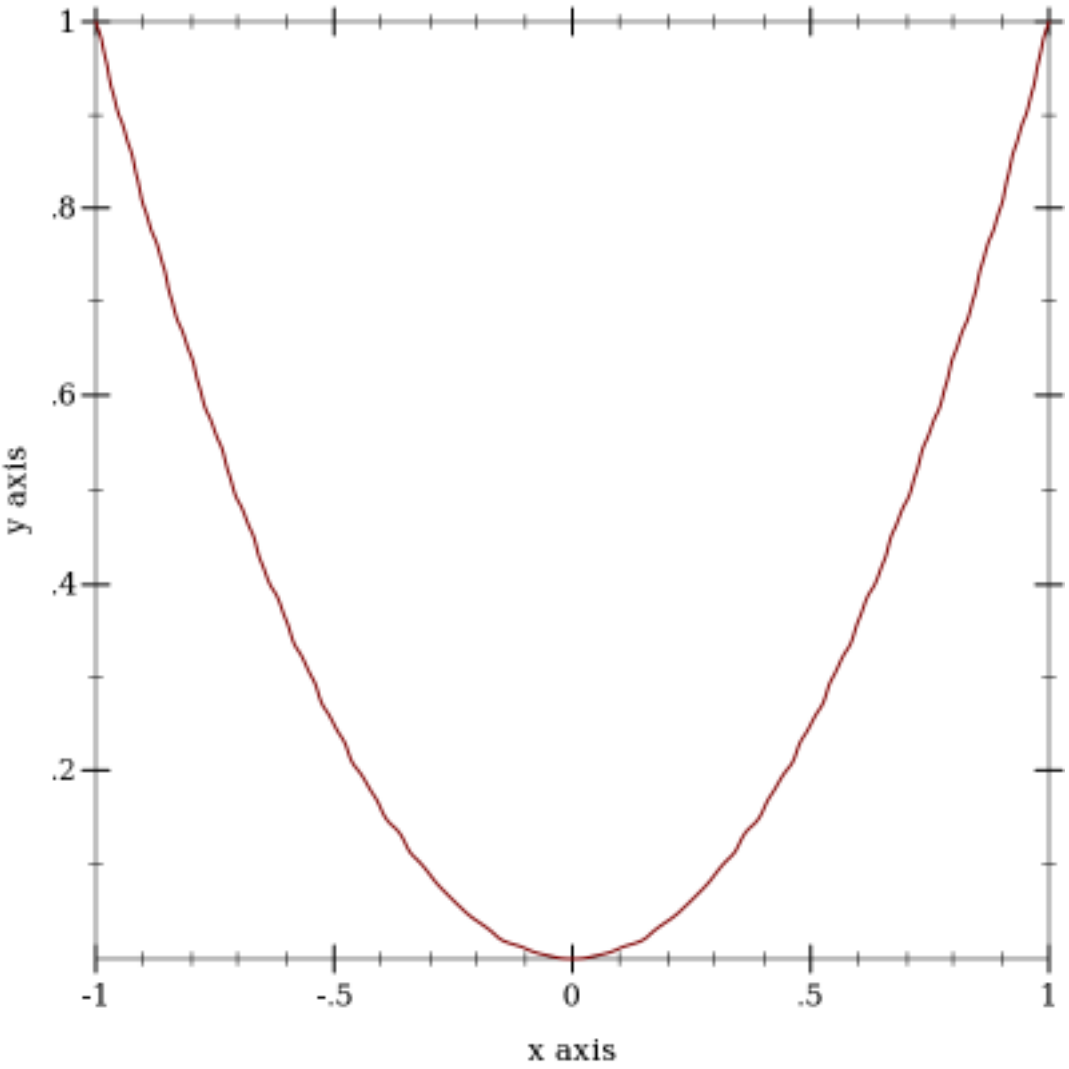
A "cube-root" transform, mostly used for testing. Unlike the log transform, it is defined on the entire real line, making it better for testing the appearance of plots with nonlinearly transformed axes.

```
(hand-drawn-transform freq) → axis-transform/c
  freq : (>/c 0)
```

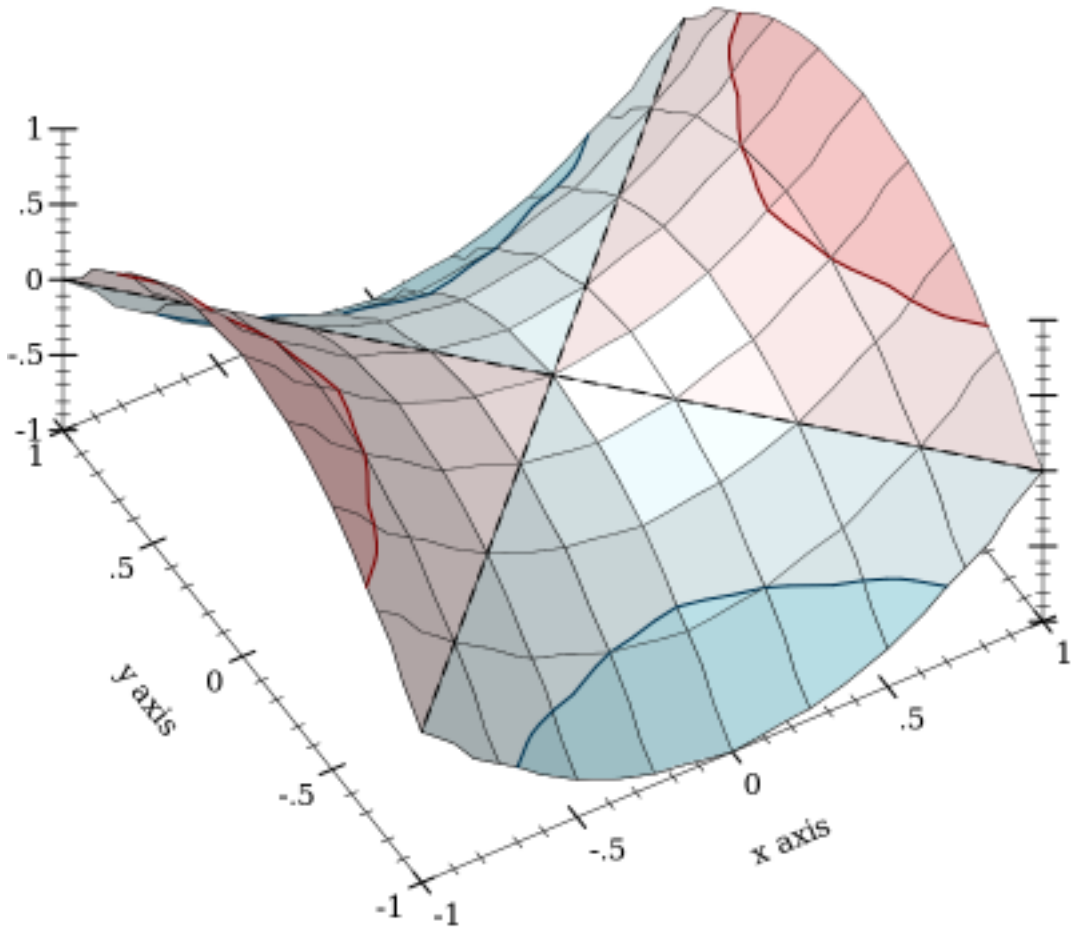
An *extremely important* test case, which makes sure that PLoT can use any monotone, invertible function as an axis transform. The `freq` parameter controls the "shakiness" of the transform. At high values, it makes plots look like Peanuts cartoons.

Examples:

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




```
> (parameterize ([plot-x-transform  (hand-drawn-transform 50)]
                 [plot-y-transform  (hand-drawn-transform 50)]
                 [plot-z-transform  (hand-drawn-transform 50)])
    (plot3d (contour-intervals3d (λ (x y) (- (sqr x) (sqr y)))
                                 -1 1 -1 1 #:samples 9)))
```




```
axis-transform/c : contract?
= (real? real? invertible-function? . -> . invertible-function?)
```

The contract for axis transforms.

The easiest ways to construct novel axis transforms are to use the axis transform combinators `axis-transform-append`, `axis-transform-bound` and `axis-transform-compose`, or to apply `make-axis-transform` to an `invertible-function`.

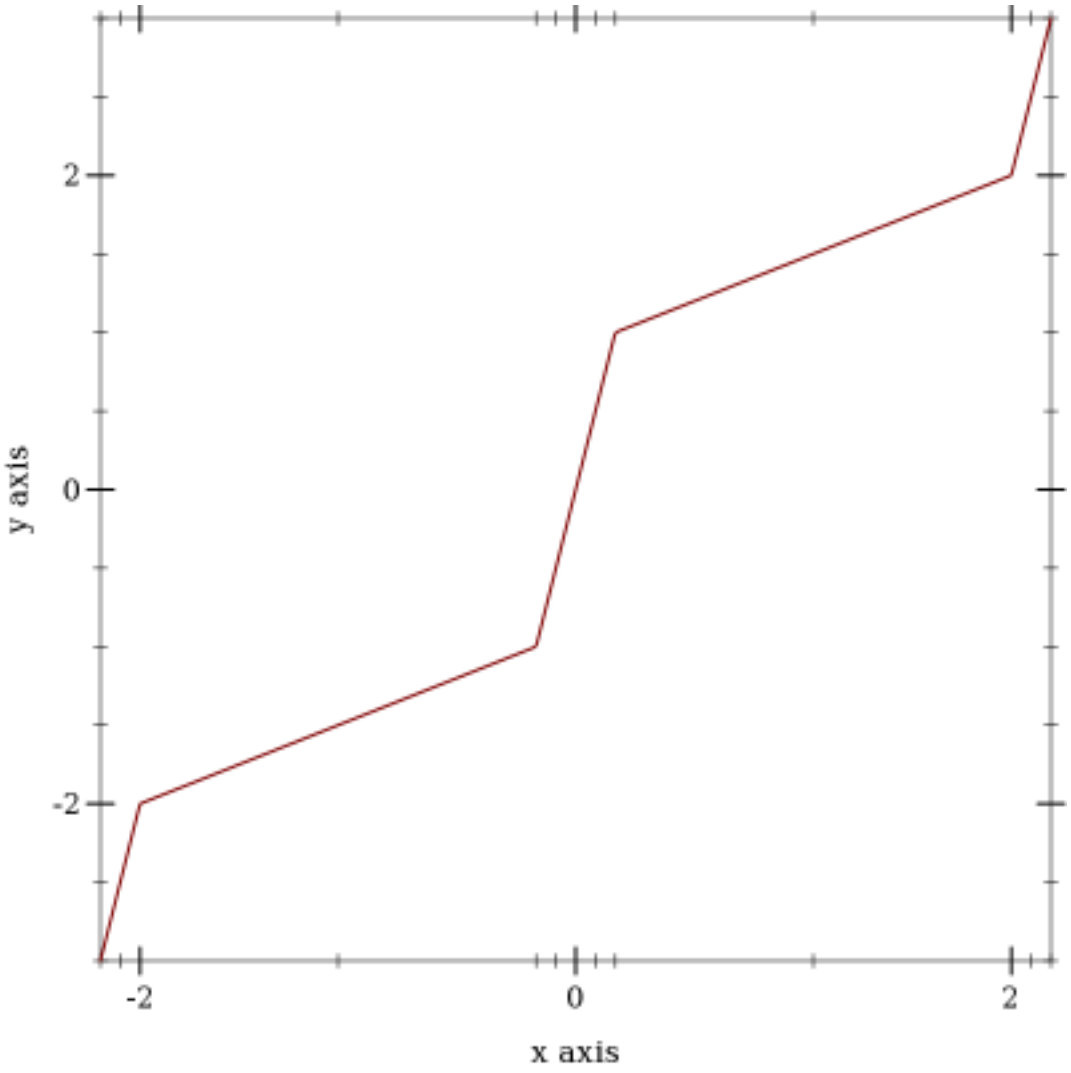
```
(axis-transform-append t1 t2 mid) → axis-transform/c
  t1 : axis-transform/c
```

```
t2 : axis-transform/c
mid : real?
```

Returns an axis transform that transforms values less than *mid* like *t1*, and transforms values greater than *mid* like *t2*. (Whether it transforms *mid* like *t1* or *t2* is immaterial, as a transformed *mid* is equal to *mid* either way.)

Example:

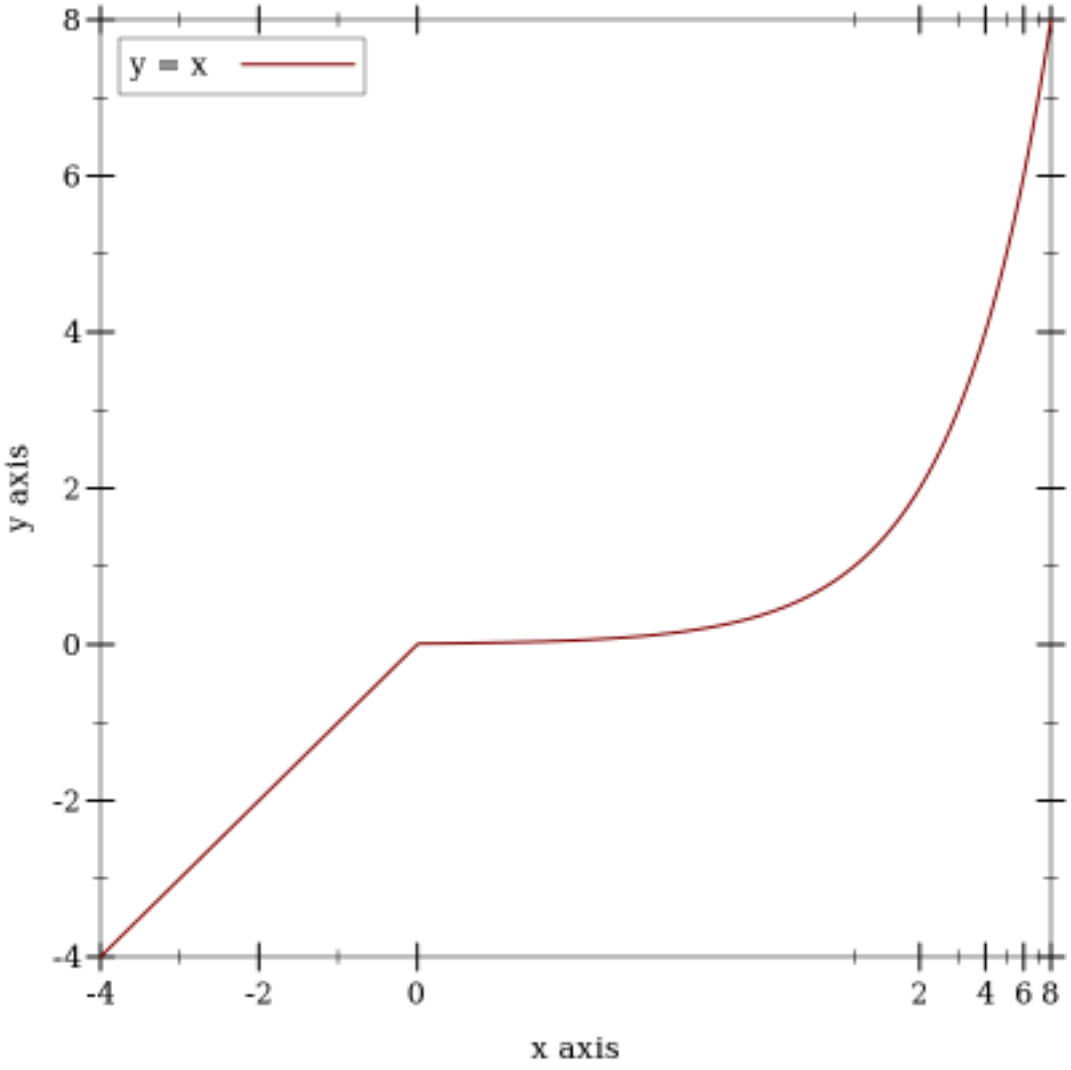
```
> (parameterize ([plot-x-transform  (axis-transform-append
                                     (stretch-transform -2 -1 10)
                                     (stretch-transform 1 2 10)
                                     0)])
    (plot (function (λ (x) x) -3 3)))
```




```
(axis-transform-bound t a b) → axis-transform/c
  t : axis-transform/c
  a : real?
  b : real?

= (axis-transform-append
    (axis-transform-append id-transform t a) id-transform b)
```

Returns an axis transform that transforms values like `t` does in the interval [`a`,`b`], but like the identity transform outside of it. For example, to bound `log-transform` to an interval in which it is well-defined,

```
> (parameterize ([plot-x-transform  (axis-transform-bound
                                     log-transform 0.01 +inf.0)])
    (plot (function (λ (x) x) -4 8 #:label "y = x")))
```

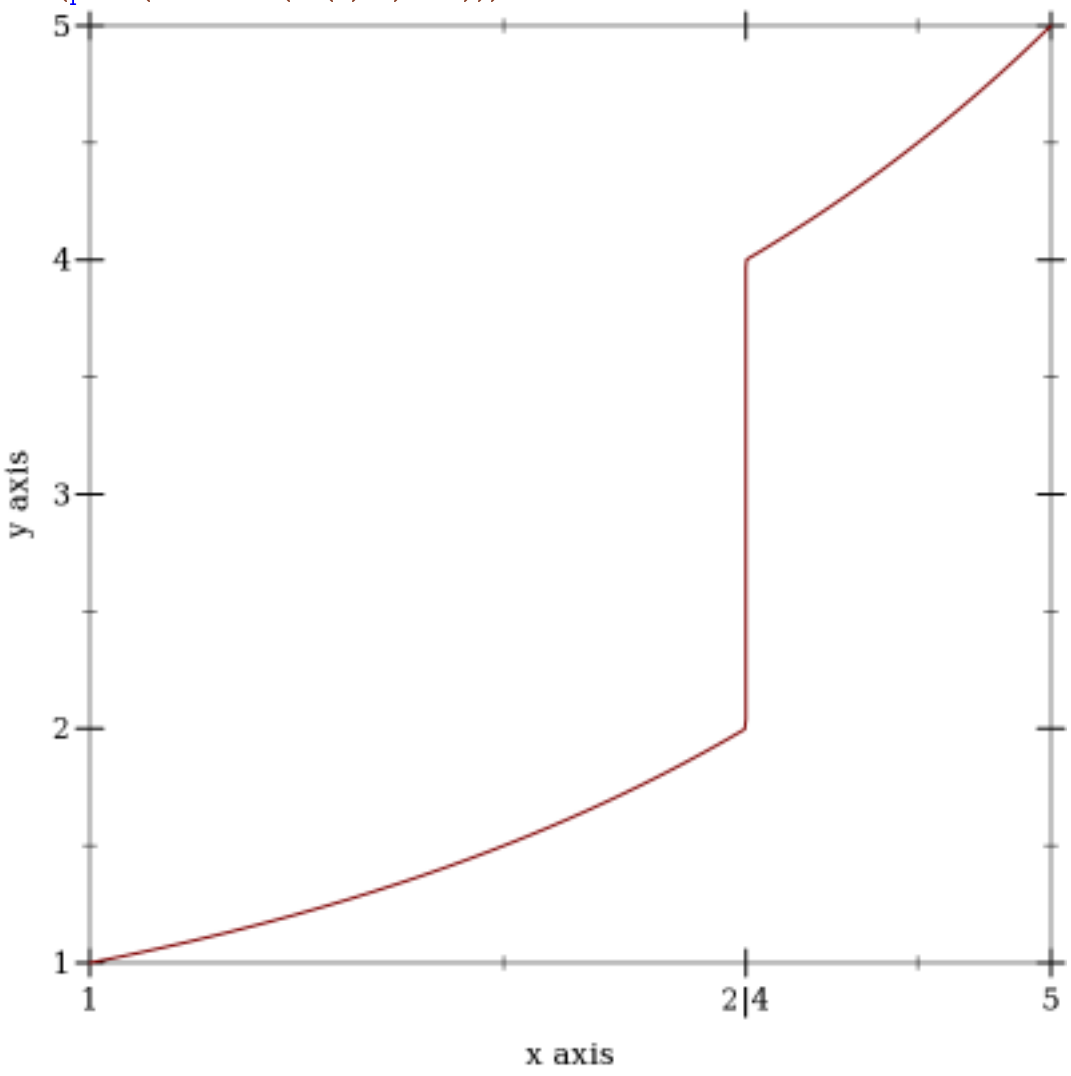



```
(axis-transform-compose t1 t2) → axis-transform/c
  t1 : axis-transform/c
  t2 : axis-transform/c
```

Composes two axis transforms. For example, to collapse part of a `log-transform`ed axis, try something like

```
> (parameterize ([plot-x-transform  (axis-transform-compose
```

```
                                    log-transform
                                    (collapse-transform 2 4))])
(plot (function (λ (x) x) 1 5)))
```



Argument order matters, but predicting the effects of exchanging arguments can be difficult. Fortunately, the effects are usually slight.

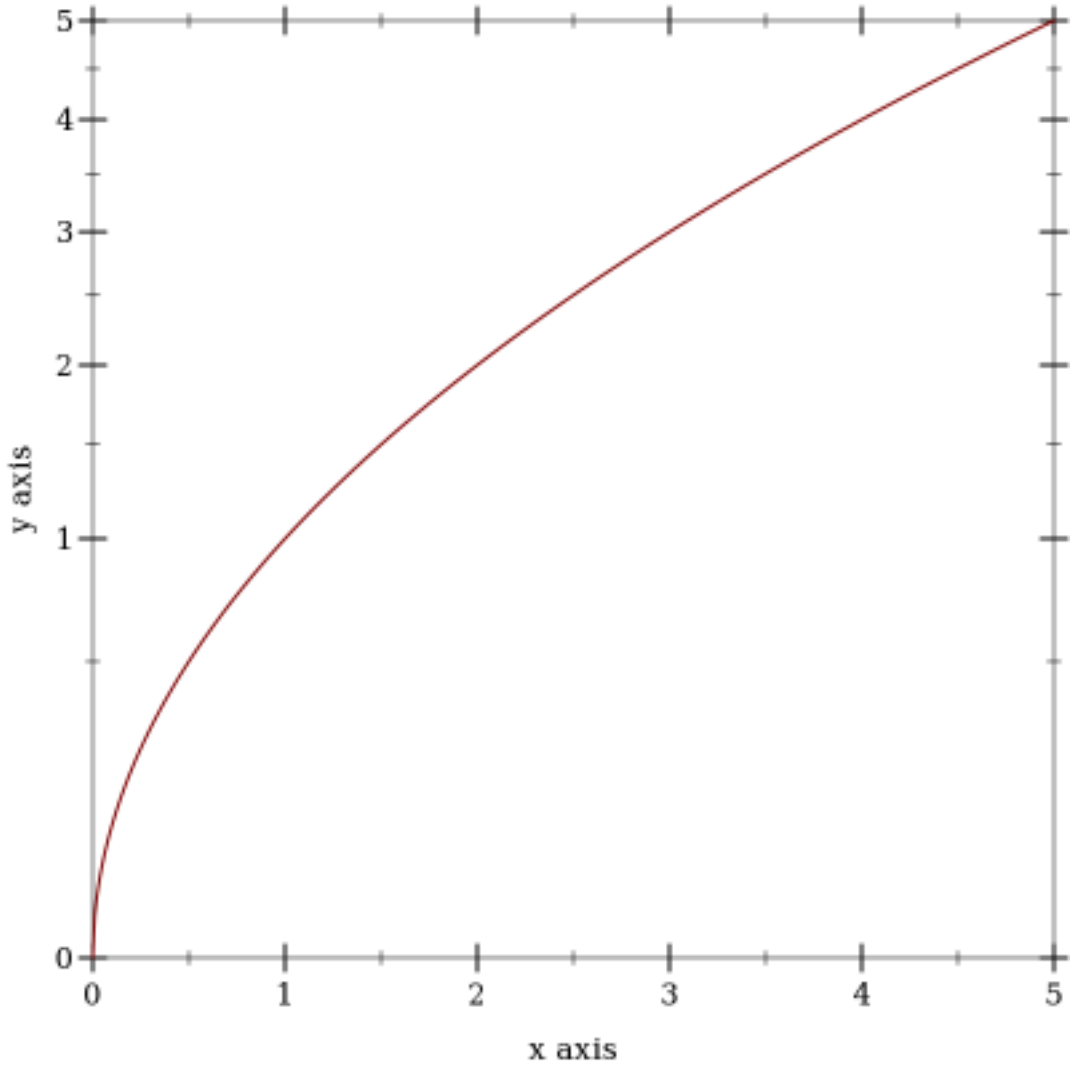
```
(make-axis-transform fun) → axis-transform/c
  fun : invertible-function?
```

Given a monotone `invertible-function`, returns an axis transform. Monotonicity is nec-

essary, but cannot be enforced. The inverse is used to take samples uniformly along transformed axes (see nonlinear-seq).

Example:

```
> (parameterize ([plot-y-transform  (make-axis-transform
                                     (invertible-
function sqrt sqr))])
    (plot (function (λ (x) x) 0 5)))
```



An axis transform created by make-axis-transform (or by any of the above combinators) does not transform the endpoints of an axis's bounds, to within floating-point error. For example,

```
> (match-let ([(invertible-function f g)
               (apply-axis-transform log-transform 1 3)])
    (define xs '(1 2 3))
    (define new-xs (map f xs))
    (define old-xs (map g new-xs))
    (values new-xs old-xs))
'(1.0 2.2618595071429146 3.0)
'(1.0 1.9999999999999998 3.0000000000000004)
```

Technically, *fun* does not need to be truly invertible. Given *fun* = `(invertible-function f g)`, it is enough for `f` to be a left inverse of `g`; that is, always `(f (g x)) = x` but not necessarily `(g (f x)) = x`. If `f` and `g` had to be strict inverses of each other, there could be no `collapse-transform`.

```
(apply-axis-transform t x-min x-max) → invertible-function?
  t : axis-transform/c
  x-min : real?
  x-max : real?

= (t x-min x-max id-function)
```

Returns an invertible function that transforms axis points within the given axis bounds. This convenience function is used internally to transform points before rendering, but is provided for completeness.

7.2 Axis Ticks

Each plot axis has two indepedent sets of ticks: the *near* ticks and the *far* ticks.

```
(plot-x-ticks) → ticks?
(plot-x-ticks ticks) → void?
  ticks : ticks?

= (linear-ticks)
```

```
(plot-x-far-ticks) → ticks?
(plot-x-far-ticks ticks) → void?
  ticks : ticks?

= (ticks-mimic plot-x-ticks)
```

```
(plot-y-ticks) → ticks?
(plot-y-ticks ticks) → void?
  ticks : ticks?

=  (linear-ticks)
```

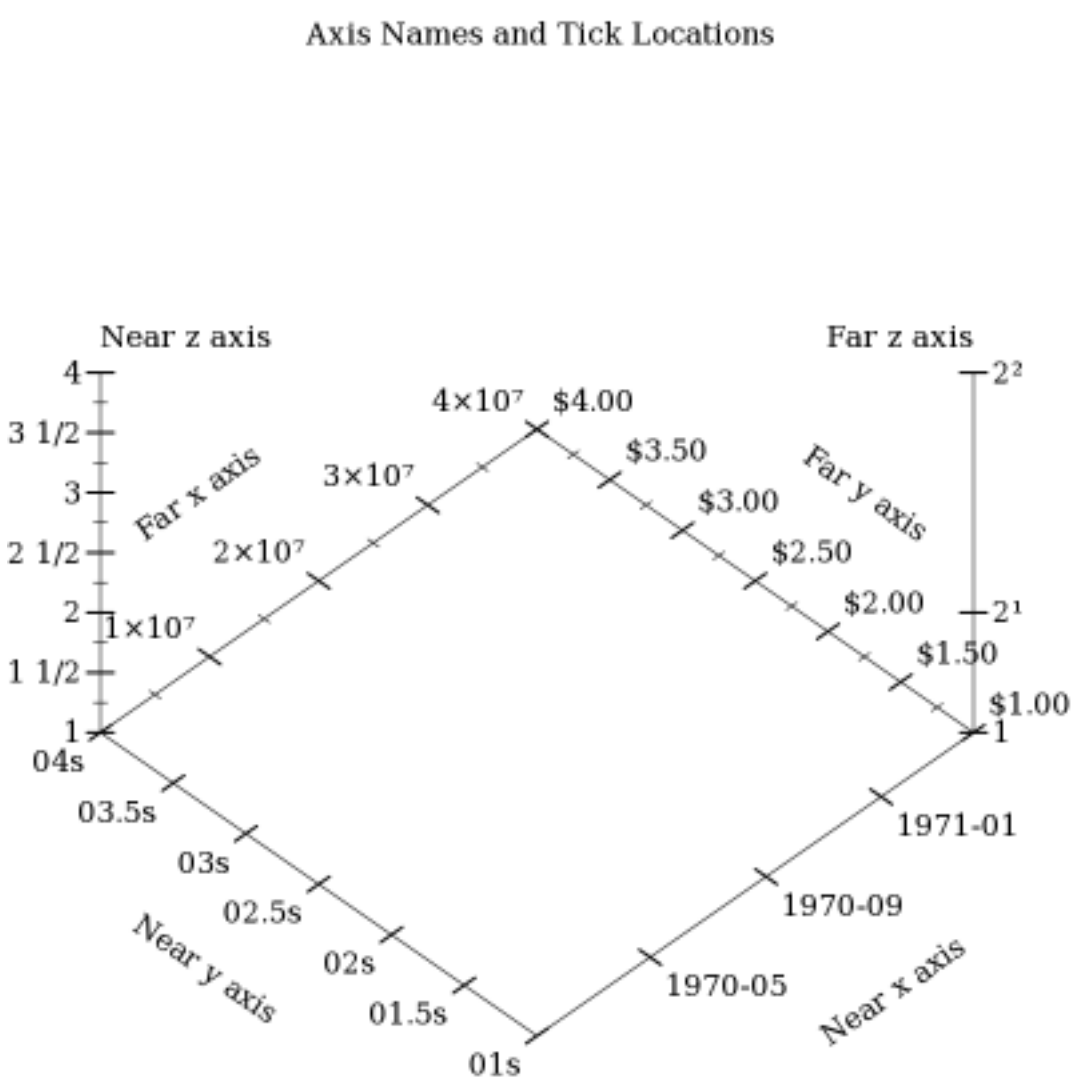
```
(plot-y-far-ticks) → ticks?
(plot-y-far-ticks ticks) → void?
  ticks : ticks?

=  (ticks-mimic plot-y-ticks)
```

```
(plot-z-ticks) → ticks?
(plot-z-ticks ticks) → void?
  ticks : ticks?

=  (linear-ticks)
```

```
(plot-z-far-ticks) → ticks?
(plot-z-far-ticks ticks) → void?
  ticks : ticks?

=  (ticks-mimic plot-z-ticks)
```

Example:

```
> (parameterize ([plot-x-label      "Near x axis"]
                 [plot-y-label      "Near y axis"]
                 [plot-z-label      "Near z axis"]
                 [plot-x-ticks      (date-ticks)]
                 [plot-y-ticks      (time-ticks)]
                 [plot-z-ticks      (fraction-ticks)]
                 [plot-x-far-label  "Far x axis"]
                 [plot-y-far-label  "Far y axis"]
                 [plot-z-far-label  "Far z axis"]
                 [plot-x-far-ticks  (linear-ticks)]
                 [plot-y-far-ticks  (currency-ticks)]
                 [plot-z-far-ticks  (log-ticks #:base 2)])
    (plot3d (lines3d '(#(1 1 1) #(40000000 4 4)) #:style 'transparent)
            #:angle 45 #:altitude 50
            #:title "Axis Names and Tick Locations"))
```



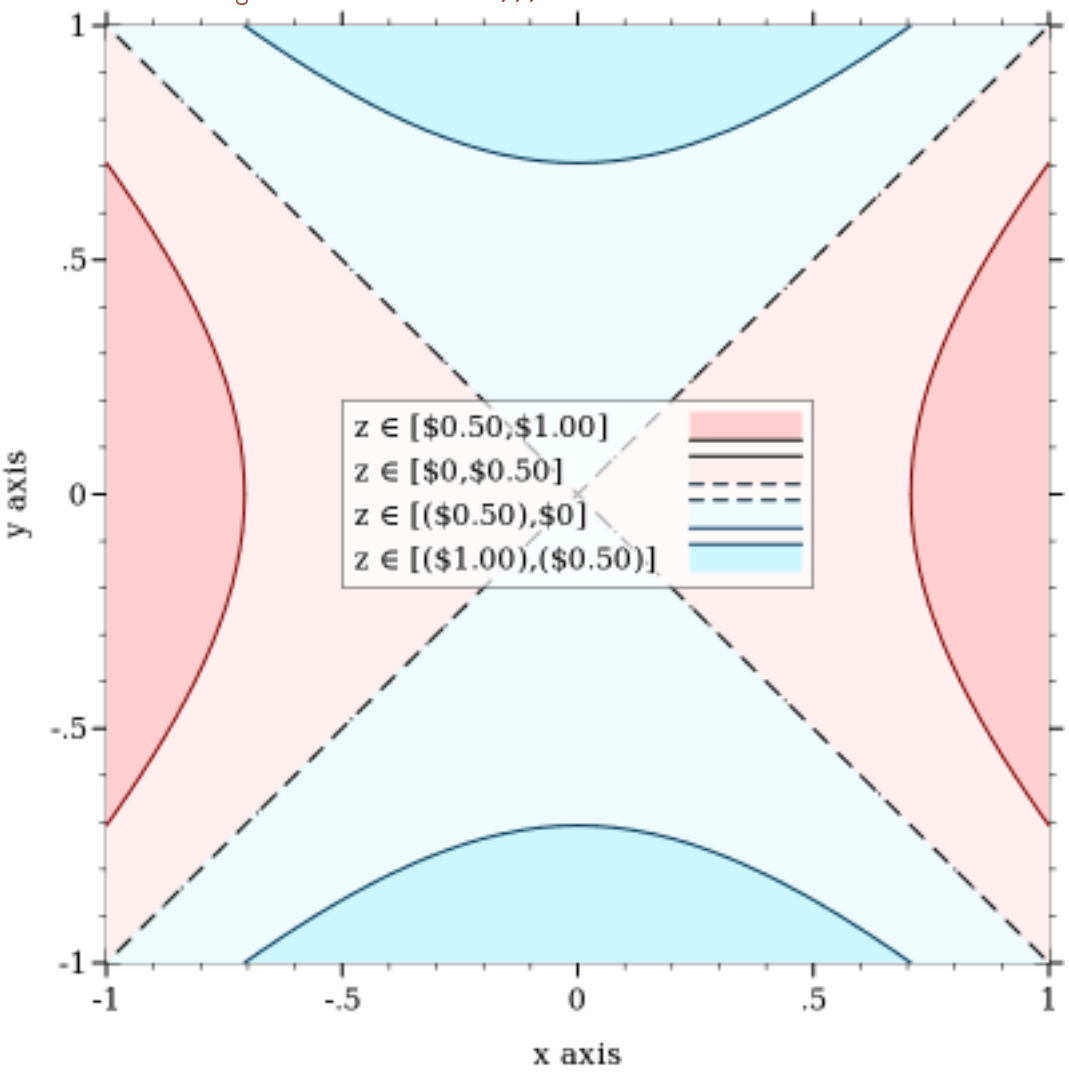
At any `#:angle`, the far *x* and *y* ticks are behind the plot, and the far *z* ticks are on the right. Far ticks are drawn, but not labeled, if they are identical to their corresponding near ticks. They are always identical by default.

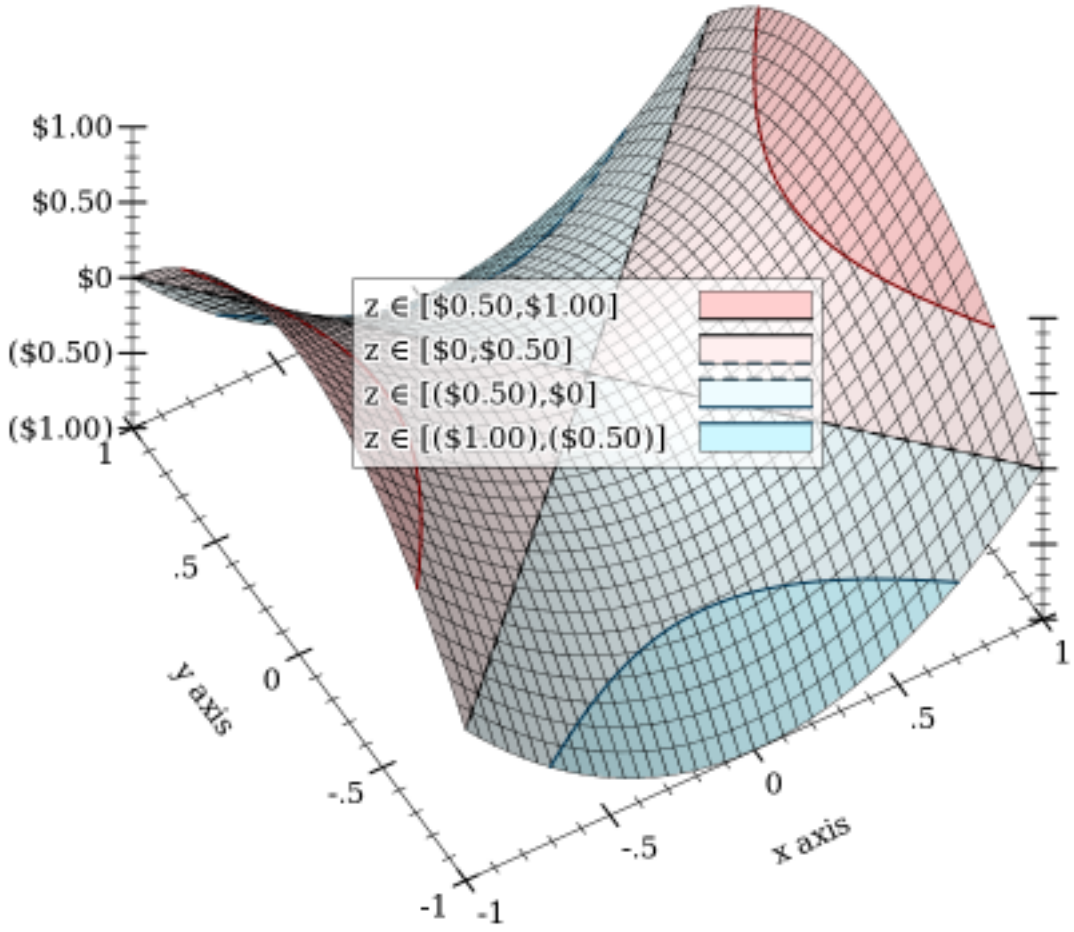
Major ticks are longer than *minor ticks*. Major tick labels are always drawn unless collapsed with a nearby tick. Minor tick labels are never drawn.

Renderers produced by `contours` and `contour-intervals` use the value of `plot-z-ticks` to place and label contour lines. For example, compare plots of the same function renderered using both `contour-intervals` and `contour-intervals3d`:

```
> (parameterize ([plot-z-ticks  (currency-ticks)])
    (define (saddle x y) (- (sqr x) (sqr y)))
```

```
(values
 (plot (contour-intervals saddle -1 1 -1 1 #:label "z")
       #:legend-anchor 'center)
 (plot3d (contour-intervals3d saddle -1 1 -1 1 #:label "z")
         #:legend-anchor 'center)))
```






```
(contour-ticks z-ticks
               z-min
               z-max
               levels
               intervals?) → (listof tick?)
  z-ticks : ticks?
  z-min : real?
  z-max : real?
  levels : (or/c 'auto exact-positive-integer? (listof real?))
  intervals? : boolean?
```

Returns the ticks used for contour values. This is used internally by renderers returned from contours, contour-intervals, contours3d, contour-intervals3d, and isosurfaces3d, but is provided for completeness.

When *levels* is 'auto, the returned values do not correspond *exactly* with the values of ticks returned by *z-ticks*: they might be missing the endpoint values. For example,

```
> (map pre-tick-value
       (filter pre-tick-major? ((plot-z-ticks) 0 1)))
'(0 1/5 2/5 3/5 4/5 1)
> (map pre-tick-value
       (contour-ticks (plot-z-ticks) 0 1 'auto #f))
'(1/5 2/5 3/5 4/5)
```

```
(plot-d-ticks) → ticks?
(plot-d-ticks ticks) → void?
  ticks : ticks?

= (linear-ticks)
```

The ticks used for default isosurface values in isosurfaces3d.

```
(plot-r-ticks) → ticks?
(plot-r-ticks ticks) → void?
  ticks : ticks?

= (linear-ticks)
```

The ticks used for radius lines in polar-axes.

```
(struct ticks (layout format)
        #:extra-constructor-name make-ticks)
  layout : ticks-layout/c
  format : ticks-format/c
```

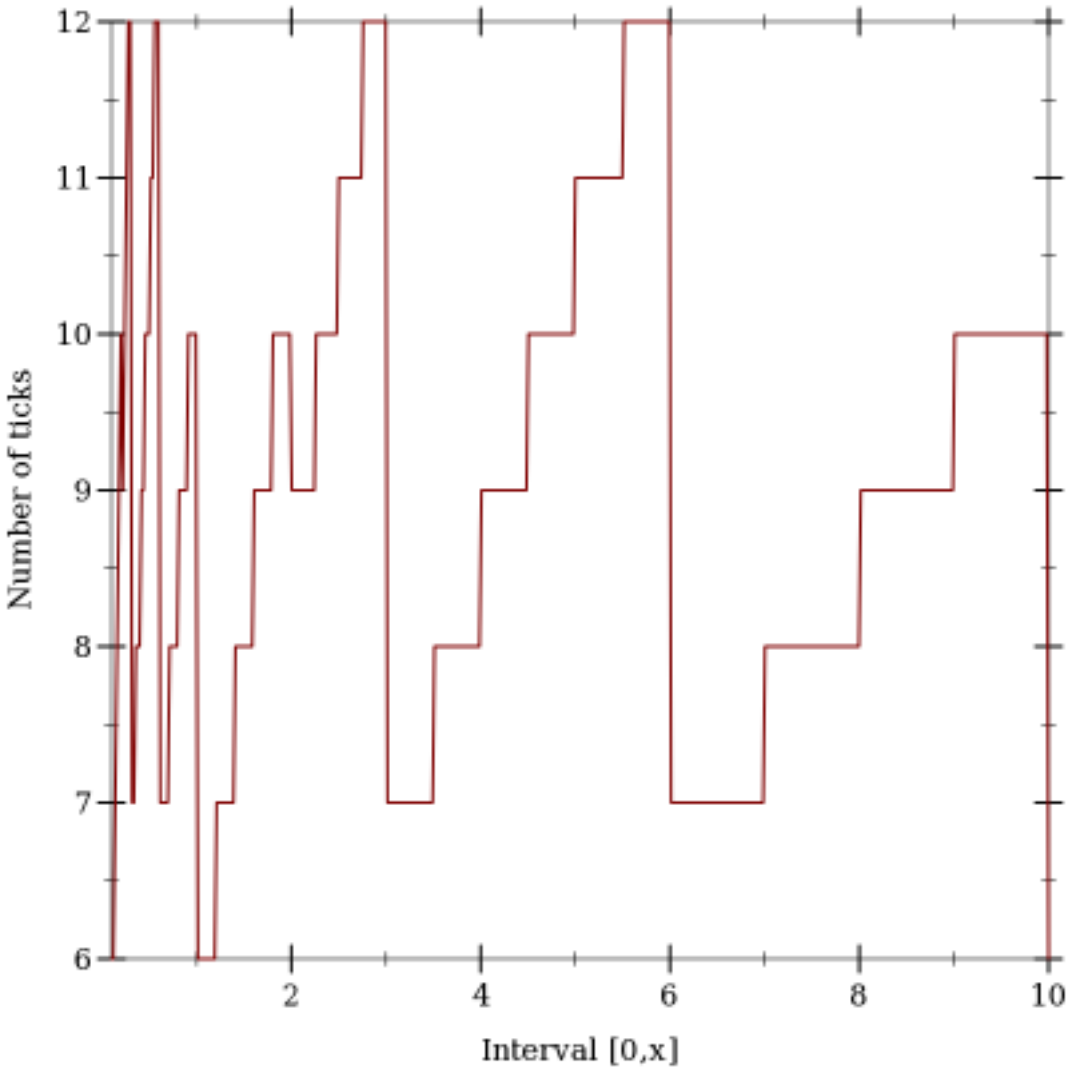
A ticks for a near or far axis consists of a layout function, which determines the number of ticks and where they will be placed, and a format function, which determines the ticks' labels.

```
(ticks-default-number) → exact-positive-integer?
(ticks-default-number number) → void?
  number : exact-positive-integer?

= 4
```

Most tick layout functions (and thus their corresponding `ticks`-constructing functions) have a `#:number` keyword argument with default `(ticks-default-number)`. What the number means depends on the tick layout function. Most use it for an average number of major ticks.

It is unlikely to mean the exact number of major ticks. Without adjusting the number of ticks, layout functions usually cannot find uniformly spaced ticks that will have simple labels after formatting. For example, the following plot shows the actual number of major ticks for the interval [0,*x*] when the requested number of ticks is 8, as generated by `linear-ticks-layout`:

```
> (plot (function (λ (x)
                    (count pre-tick-major?
                           ((linear-ticks-
layout #:number 8) 0 x)))
                  0.1 10)
        #:x-label "Interval [0,x]" #:y-label "Number of ticks")
```



7.2.1 Linear Ticks

```
(linear-ticks-layout [#:number number
                      #:base base
                      #:divisors divisors]) → ticks-layout/c
  number : exact-positive-integer? = (ticks-default-number)
  base : (and/c exact-integer? (>=/c 2)) = 10
  divisors : (listof exact-positive-integer?) = '(1 2 4 5)
```

```
(linear-ticks-format) → ticks-format/c
```

```
(linear-ticks [#:number number
               #:base base
               #:divisors divisors]) → ticks?
  number : exact-positive-integer? = (ticks-default-number)
  base : (and/c exact-integer? (>=/c 2)) = 10
  divisors : (listof exact-positive-integer?) = '(1 2 4 5)

=  (ticks (linear-ticks-layout #:number number #:base base
                               #:divisors divisors)
          (linear-ticks-format))
```

The layout function, format function, and combined `ticks` for uniformly spaced ticks.

To lay out ticks, `linear-ticks-layout` finds the power of `base` closest to the axis interval size, chooses a simple first tick, and then chooses a skip length using `divisors` that maximizes the number of ticks without exceeding `number`. The default arguments correspond to the standard 1-2-5-in-base-10 rule used almost everywhere in plot tick layout.

To format ticks, `linear-ticks-format` uses `real->plot-label`, and uses `digits-for-range` to determine the maximum number of fractional digits in the decimal expansion.

For strategic use of non-default arguments, see `bit/byte-ticks`, `currency-ticks`, and `fraction-ticks`.

7.2.2 Log Ticks

```
(log-ticks-layout [#:number number
                   #:base base])    → ticks-layout/c
  number : exact-positive-integer? = (ticks-default-number)
  base : (and/c exact-integer? (>=/c 2)) = 10
```

```
(log-ticks-format [#:base base]) → ticks-format/c
  base : (and/c exact-integer? (>=/c 2)) = 10
```

```
(log-ticks [#:number number #:base base]) → ticks?
  number : exact-positive-integer? = (ticks-default-number)
  base : (and/c exact-integer? (>=/c 2)) = 10

=  (ticks (log-ticks-layout #:number number #:base base)
          (log-ticks-format #:base base))
```

The layout function, format function, and combined `ticks` for exponentially spaced major ticks. (The minor ticks between are uniformly spaced.) Use these ticks for

log-transformed axes, because when exponentially spaced tick positions are log-transformed, they become uniformly spaced.

The #:base keyword argument is the logarithm base. See plot-z-far-ticks for an example of use.

7.2.3 Date Ticks

```
(date-ticks-layout [#:number number]) → ticks-layout/c
  number : exact-positive-integer? = (ticks-default-number)
```

```
(date-ticks-format [#:formats formats]) → ticks-format/c
  formats : (listof string?) = (date-ticks-formats)
```

```
(date-ticks [#:number number
             #:formats formats]) → ticks?
  number : exact-positive-integer? = (ticks-default-number)
  formats : (listof string?) = (date-ticks-formats)

=  (ticks (date-ticks-layout #:number number)
          (date-ticks-format #:formats formats))
```

The layout function, format function, and combined ticks for uniformly spaced ticks with date labels.

These axis ticks regard values as being in seconds since *a system-dependent Universal Coordinated Time (UTC) epoch*. (For example, the Unix and Mac OS X epoch is January 1, 1970 UTC, and the Windows epoch is January 1, 1601 UTC.) Use date->seconds to convert local dates to seconds, or datetime->real to convert dates to UTC seconds in a way that accounts for time zone offsets.

Actually, date-ticks-layout does not always space ticks *quite* uniformly. For example, it rounds ticks that are spaced about one month apart or more to the nearest month. Generally, date-ticks-layout tries to place ticks at minute, hour, day, week, month and year boundaries, as well as common multiples such as 90 days or 6 months.

To try to avoid displaying overlapping labels, date-ticks-format chooses date formats from formats for which labels will contain no redundant information.

All the format specifiers given in srfi/19 (which are derived from Unix's date command), except those that represent time zones, are allowed in date format strings.

```
(date-ticks-formats) → (listof string?)
(date-ticks-formats formats) → void?
  formats : (listof string?)
```

```
= 24h-descending-date-ticks-formats
```

The default date formats.

```
24h-descending-date-ticks-formats : (listof string?)

= '("~Y-~m-~d ~H:~M:~f"
    "~Y-~m-~d ~H:~M"
    "~Y-~m-~d ~Hh"
    "~Y-~m-~d"
    "~Y-~m"
    "~Y"
    "~m-~d ~H:~M:~f"
    "~m-~d ~H:~M"
    "~m-~d ~Hh"
    "~m-~d"
    "~H:~M:~f"
    "~H:~M"
    "~Hh"
    "~M:~fs"
    "~Mm"
    "~fs")
```

```
12h-descending-date-ticks-formats : (listof string?)

= '("~Y-~m-~d ~I:~M:~f ~p"
    "~Y-~m-~d ~I:~M ~p"
    "~Y-~m-~d ~I ~p"
    "~Y-~m-~d"
    "~Y-~m"
    "~Y"
    "~m-~d ~I:~M:~f ~p"
    "~m-~d ~I:~M ~p"
    "~m-~d ~I ~p"
    "~m-~d"
    "~I:~M:~f ~p"
    "~I:~M ~p"
    "~I ~p"
    "~M:~fs"
    "~Mm"
    "~fs")
```

7.2.4 Time Ticks

```
(time-ticks-layout [#:number number]) → ticks-layout/c
  number : exact-positive-integer? = (ticks-default-number)
```

```
(time-ticks-format [#:formats formats]) → ticks-format/c
  formats : (listof string?) = (time-ticks-formats)
```

```
(time-ticks [#:number number
             #:formats formats]) → ticks?
  number : exact-positive-integer? = (ticks-default-number)
  formats : (listof string?) = (time-ticks-formats)

=  (ticks (time-ticks-layout #:number number)
          (time-ticks-format #:formats formats))
```

The layout function, format function, and combined `ticks` for uniformly spaced ticks with time labels.

These axis ticks regard values as being in seconds. Use `datetime->real` to convert `sql-time` or `plot-time` values to seconds.

Generally, `time-ticks-layout` tries to place ticks at minute, hour and day boundaries, as well as common multiples such as 12 hours or 30 days.

To try to avoid displaying overlapping labels, `time-ticks-format` chooses a date format from `formats` for which labels will contain no redundant information.

All the time-related format specifiers given in `srfi/19` (which are derived from Unix's `date` command) are allowed in time format strings.

```
(time-ticks-formats) → (listof string?)
(time-ticks-formats formats) → void?
  formats : (listof string?)

=  24h-descending-time-ticks-formats
```

The default time formats.

```
24h-descending-time-ticks-formats : (listof string?)
```

```
=  '("~dd ~H:~M:~f"
     "~dd ~H:~M"
     "~dd ~Hh"
     "~dd"
     "~H:~M:~f"
     "~H:~M"
     "~Hh"
     "~M:~fs"
     "~Mm"
     "~fs")
```

```
12h-descending-time-ticks-formats : (listof string?)

=  '("~dd ~I:~M:~f ~p"
     "~dd ~I:~M ~p"
     "~dd ~I ~p"
     "~dd"
     "~I:~M:~f ~p"
     "~I:~M ~p"
     "~I ~p"
     "~M:~fs"
     "~Mm"
     "~fs")
```

7.2.5 Currency Ticks

```
(currency-ticks-format [#:kind kind
                        #:scales scales
                        #:formats formats]) → ticks-format/c
  kind : (or/c string? symbol?) = 'USD
  scales : (listof string?) = (currency-ticks-scales)
  formats : (list/c string? string? string?)
          = (currency-ticks-formats)
```

```
(currency-ticks [#:number number
                 #:kind kind
                 #:scales scales
                 #:formats formats]) → ticks?
  number : exact-positive-integer? = (ticks-default-number)
  kind : (or/c string? symbol?) = 'USD
  scales : (listof string?) = (currency-ticks-scales)
  formats : (list/c string? string? string?)
          = (currency-ticks-formats)

=  (ticks (linear-ticks-layout #:number number)
          (currency-ticks-format #:kind kind #:scales scales
                                 #:formats formats))
```

The format function and combined `ticks` for uniformly spaced ticks with currency labels.

The `#:kind` keyword argument is either a string containing the currency symbol, or a currency code such as `'USD`, `'GBP` or `'EUR`. The `currency-ticks-format` function can map most ISO 4217 currency codes to their corresponding currency symbol.

The `#:scales` keyword argument is a list of suffixes for each 10^3 scale, such as `"K"` (US thousand, or kilo), `"bn"` (UK short-scale billion) or `"Md"` (EU long-scale milliard). Off-scale amounts are given power-of-ten suffixes such as "$\times 10^{21}$."

The `#:formats` keyword argument is a list of three format strings, representing the formats of positive, negative, and zero amounts, respectively. The format specifiers are:

- `"~$"`: replaced by the currency symbol
- `"~w"`: replaced by the whole part of the amount
- `"~f"`: replaced by the fractional part, with 2 or more decimal digits
- `"~s"`: replaced by the scale suffix
- `"~~"`: replaced by "~"

```
(currency-ticks-scales) → (listof string?)
(currency-ticks-scales scales) → void?
  scales : (listof string?)

=  us-currency-scales
```

```
(currency-ticks-formats) → (list/c string? string? string?)
(currency-ticks-formats formats) → void?
  formats : (list/c string? string? string?)

=  us-currency-formats
```

The default currency scales and formats.

For example, a PLoT user in France would probably begin programs with

```
(require plot)
(currency-ticks-scales eu-currency-scales)
(currency-ticks-formats eu-currency-formats)
```

and use `(currency-ticks #:kind 'EUR)` for local currency or `(currency-ticks #:kind 'JPY)` for Japanese Yen.

Cultural sensitivity notwithstanding, when writing for a local audience, it is generally considered proper to use local currency scales and formats for foreign currencies, but use the foreign currency symbol.

```
us-currency-scales : (listof string?)
= '("" "K" "M" "B" "T")
```

Short-scale suffix abbreviations as commonly used in the United States, Canada, and some other English-speaking countries. These stand for “kilo,” “million,” “billion,” and “trillion.”

```
uk-currency-scales : (listof string?)
= '("" "k" "m" "bn" "tr")
```

Short-scale suffix abbreviations as commonly used in the United Kingdom since switching to the short scale in 1974, and as currently recommended by the Daily Telegraph and Times style guides.

```
eu-currency-scales : (listof string?)
= '("" "K" "M" "Md" "B")
```

European Union long-scale suffix abbreviations, which stand for “kilo,” “million,” “milliard,” and “billion.”

The abbreviations actually used vary with geography, even within countries, but these seem to be common. Further long-scale suffix abbreviations such as for “billiard” are ommitted due to lack of even weak consensus.

```
us-currency-formats : (list/c string? string? string?)
= '("~$~w.~f~s" "(~$~w.~f~s)" "~$0")
```

Common currency formats used in the United States.

```
uk-currency-formats : (list/c string? string? string?)
= '("~$~w.~f~s" "-~$~w.~f~s" "~$0")
```

Common currency formats used in the United Kingdom. Note that it sensibly uses a negative sign to denote negative amounts.

```
eu-currency-formats : (list/c string? string? string?)
= '("~w,~f ~s~$" "-~w,~f ~s~$" "0 ~$")
```

A guess at common currency formats for the European Union. Like scale suffixes, actual formats vary with geography, but currency formats can even vary with audience or tone.

7.2.6 Other Ticks

```
no-ticks-layout : ticks-layout/c
= (λ (x-min x-max) empty)
```

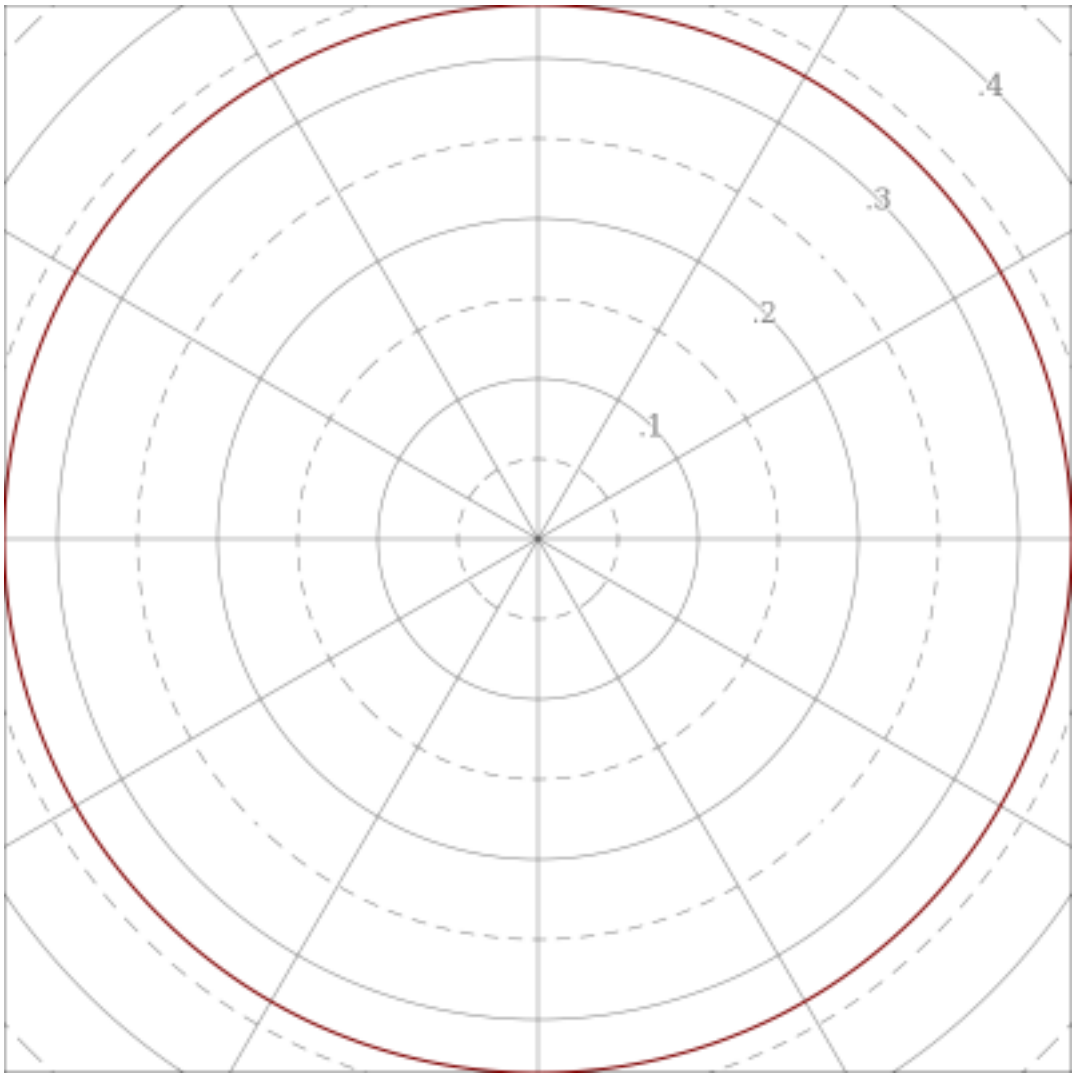
```
no-ticks-format : ticks-format/c
= (λ (x-min x-max pre-ticks)
    (map (λ (_) "") pre-ticks))
```

```
no-ticks : ticks?
= (ticks no-ticks-layout no-ticks-format)
```

The layout function, format function, and combined `ticks` for no ticks whatsoever.

Example:

```
> (parameterize ([plot-x-ticks  no-ticks]
                 [plot-y-ticks  no-ticks]
                 [plot-x-label  #f]
                 [plot-y-label  #f])
    (plot (list (polar-axes) (polar (λ (θ) 1/3)))))
```




```
(bit/byte-ticks-format [#:size size
                        #:kind kind]) → ticks-format/c
  size : (or/c 'byte 'bit) = 'byte
  kind : (or/c 'CS 'SI) = 'CS
```

```
(bit/byte-ticks [#:number number
                 #:size size
                 #:kind kind])    → ticks?
  number : exact-positive-integer? = (ticks-default-number)
  size : (or/c 'byte 'bit) = 'byte
  kind : (or/c 'CS 'SI) = 'CS
```

```
= (define si? (eq? kind 'SI))
  (ticks (linear-ticks-layout #:number number #:base (if si? 10 2)
                              #:divisors (if si? '(1 2 4 5) '(1 2)))
         (bit/byte-ticks-format #:size size #:kind kind))
```

The format function and and combined `ticks` for bit or byte values.

The `#:kind` keyword argument indicates either International System of Units (`'SI`) suffixes, as used to communicate hard drive capacities, or Computer Science (`'CS`) suffixes, as used to communicate memory capacities.

```
(fraction-ticks-format [#:base base
                        #:divisors divisors]) → ticks-format/c
  base : (and/c exact-integer? (>=/c 2)) = 10
  divisors : (listof exact-positive-integer?) = '(1 2 3 4 5)
```

```
(fraction-ticks [#:base base
                 #:divisors divisors]) → ticks?
  base : (and/c exact-integer? (>=/c 2)) = 10
  divisors : (listof exact-positive-integer?) = '(1 2 3 4 5)

= (ticks (linear-ticks #:base base #:divisors divisors)
         (fraction-ticks-format #:base base #:divisors divisors))
```

The format function and and combined `ticks` for fraction-formatted values.

7.2.7 Tick Combinators

```
(ticks-mimic thunk) → ticks?
  thunk : (-> ticks?)
```

Returns a `ticks` that mimics the given `ticks` returned by *`thunk`*. Used in default values for `plot-x-far-ticks`, `plot-y-far-ticks` and `plot-z-far-ticks` to ensure that, unless one of these parameters is changed, the far tick labels are not drawn.

```
(ticks-add t xs [major?]) → ticks?
  t : ticks?
  xs : (listof real?)
  major? : boolean? = #t
```

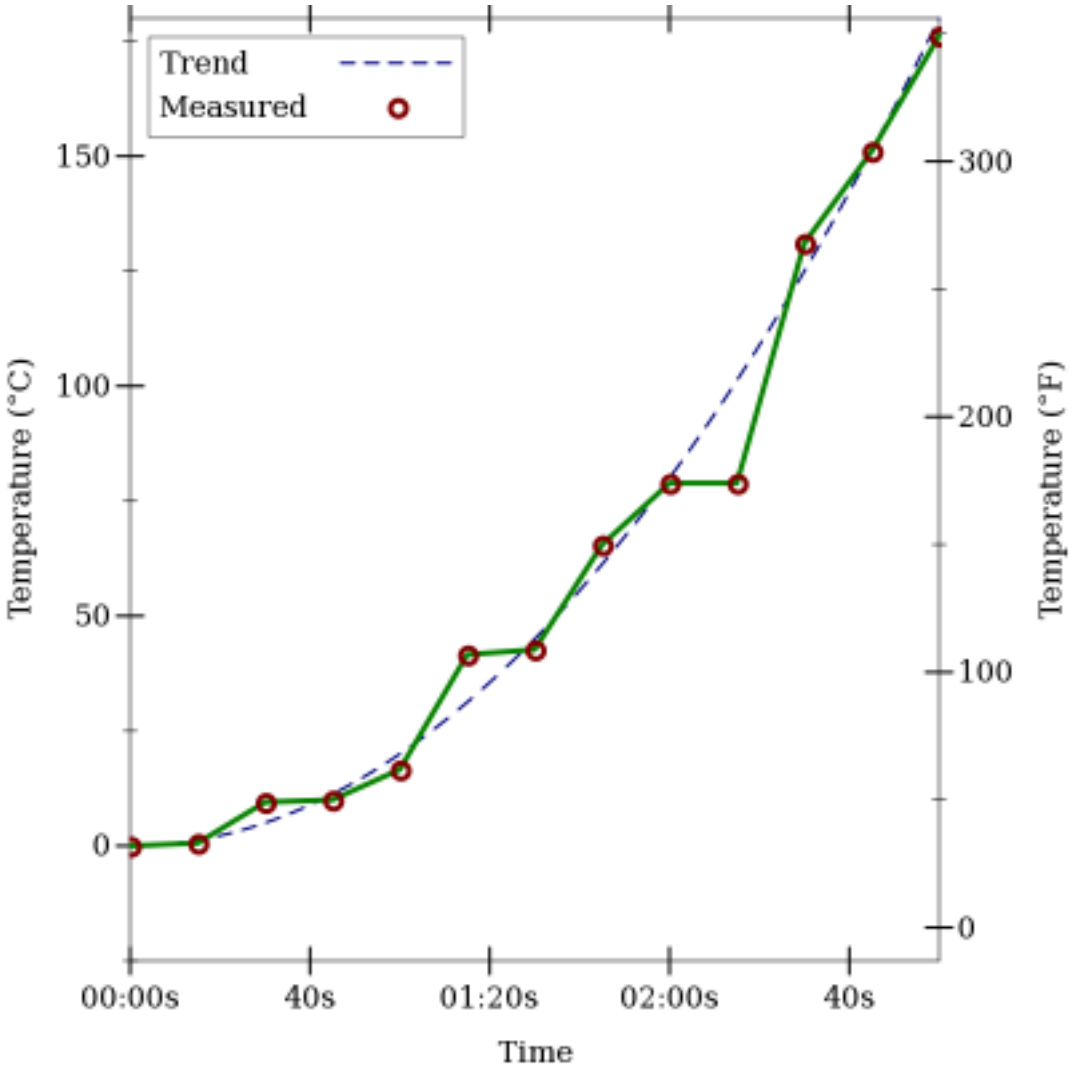
Returns a new `ticks` that acts like `t`, except that it puts additional ticks at positions *`xs`*. If *`major?`* is true, the ticks at positions *`xs`* are all major ticks; otherwise, they are minor ticks.

```
(ticks-scale t fun) → ticks?
  t : ticks?
  fun : invertible-function?
```

Returns a new `ticks` that acts like `t`, but for an axis transformed by *`fun`*. Unlike with typical §7.1 “Axis Transforms”, *`fun`* is allowed to transform axis endpoints. (See `make-axis-transform` for an explanation about transforming endpoints.)

Use `ticks-scale` to plot values at multiple scales simultaneously, with one scale on the near axis and one scale on the far axis. The following example plots degrees Celsius on the left and degrees Farenheit on the right:

```
> (parameterize
      ([plot-x-ticks      (time-ticks)]
       [plot-y-far-ticks  (ticks-scale (plot-y-ticks)
                                       (linear-scale 9/5 32))]
       [plot-y-label      "Temperature (°C)"]
       [plot-y-far-label  "Temperature (°F)"])
    (define data
      (list #(0 0) #(15 0.6) #(30 9.5) #(45 10.0) #(60 16.6)
            #(75 41.6) #(90 42.7) #(105 65.5) #(120 78.9)
            #(135 78.9) #(150 131.1) #(165 151.1) #(180 176.2)))
    (plot (list
           (function (λ (x) (/ (sqr x) 180)) 0 180
                     #:style 'long-dash #:color 3 #:label "Trend")
           (lines data #:color 2 #:width 2)
           (points data #:color 1 #:line-width 2 #:label "Measured"))
          #:y-min -25 #:x-label "Time"))
```



7.2.8 Tick Data Types and Contracts

```
(struct pre-tick (value major?)
        #:extra-constructor-name make-pre-tick)
  value : real?
  major? : boolean?
```

Represents a tick that has not yet been labeled.

```
(struct tick pre-tick (label)
        #:extra-constructor-name make-tick)
  label : string?
```

Represents a tick with a label.

```
ticks-layout/c : contract?
=  (real? real? . -> . (listof pre-tick?))
```

The contract for tick layout functions. Note that a layout function returns `pre-tick`s, or unlabeled ticks.

```
ticks-format/c : contract?
=  (real? real? (listof pre-tick?) . -> . (listof string?))
```

The contract for tick format functions. A format function receives axis bounds so it can determine how many decimal digits to display (usually by applying `digits-for-range` to the bounds).

7.3 Invertible Functions

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

Represents an invertible function. Used for §7.1 “Axis Transforms” and by `ticks-scale`.

The function itself is `f`, and its inverse is `g`. 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.

```
id-function : invertible-function?
```

The identity function as an `invertible-function`.

```
(invertible-compose f1 f2) → invertible-function?
  f1 : invertible-function?
  f2 : invertible-function?
```

Returns the composition of two invertible functions.

```
(invertible-inverse h) → invertible-function?
  h : invertible-function?

= (match-define (invertible-function f g) h)
  (invertible-function g f)
```

Returns the inverse of an invertible function.

```
(linear-scale m [b]) → invertible-function?
  m : rational?
  b : rational? = 0

= (invertible-function (λ (x) (+ (* m x) b))
                       (λ (y) (/ (- y b) m)))
```

Returns a one-dimensional linear scaling function, as an `invertible-function`. This function constructs the most common arguments to `ticks-scale`.

8 Plot Utilities

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

8.1 Formatting

```
(digits-for-range  x-min
                   x-max
                  [base
                   extra-digits]) → exact-integer?
  x-min : real?
  x-max : real?
  base : (and/c exact-integer? (>=/c 2)) = 10
  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 [scientific?]) → any
  x : real?
  digits : exact-integer?
  scientific? : boolean? = #t
```

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)
"1×10⁹"
> (real->plot-label 1000000000.1234 4)
"(1×10⁹)+.1234"
```

```
(ivl->plot-label i [extra-digits]) → string?
  i : ivl?
  extra-digits : exact-integer? = 3
```

Converts an interval to a plot label.

If `i` = `(ivl x-min x-max)`, the number of digits used is `(digits-for-range x-min x-max 10 extra-digits)` when both endpoints are `rational?`. Otherwise, it is unspecified—but will probably remain `15`.

Examples:

```
> (ivl->plot-label (ivl -10.52312 10.99232))
"[-10.52,10.99]"
> (ivl->plot-label (ivl -inf.0 pi))
"[-inf.0,3.141592653589793]"
```

```
(->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 any trailing zeros and any trailing decimal point.

```
(real->decimal-string* x
                       min-digits
                       [max-digits]) → string?
  x : real?
  min-digits : exact-nonnegative-integer?
  max-digits : exact-nonnegative-integer? = min-digits
```

Like `real->decimal-string`, but accepts both a maximum and minimum number of digits.

Examples:

```
> (real->decimal-string* 1 5 10)
"1.00000"
```

```
> (real->decimal-string* 1.123456 5 10)
"1.123456"
> (real->decimal-string* 1.123456789123456 5 10)
"1.1234567891"
```

Applying `(real->decimal-string* x min-digits)` yields the same value as `(real->decimal-string x min-digits)`.

```
(integer->superscript x) → string?
  x : exact-integer?
```

Converts an integer into a string of superscript Unicode characters.

Example:

```
> (integer->superscript -1234567890)
"⁻¹²³⁴⁵⁶⁷⁸⁹⁰"
```

Systems running some out-of-date versions of Windows XP have difficulty with Unicode superscripts for 4 and up. Because `integer->superscript` is used by every number formatting function to format exponents, if you have such a system, PLoT will apparently not format all numbers with exponents correctly (until you update it).

8.2 Sampling

```
(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
```

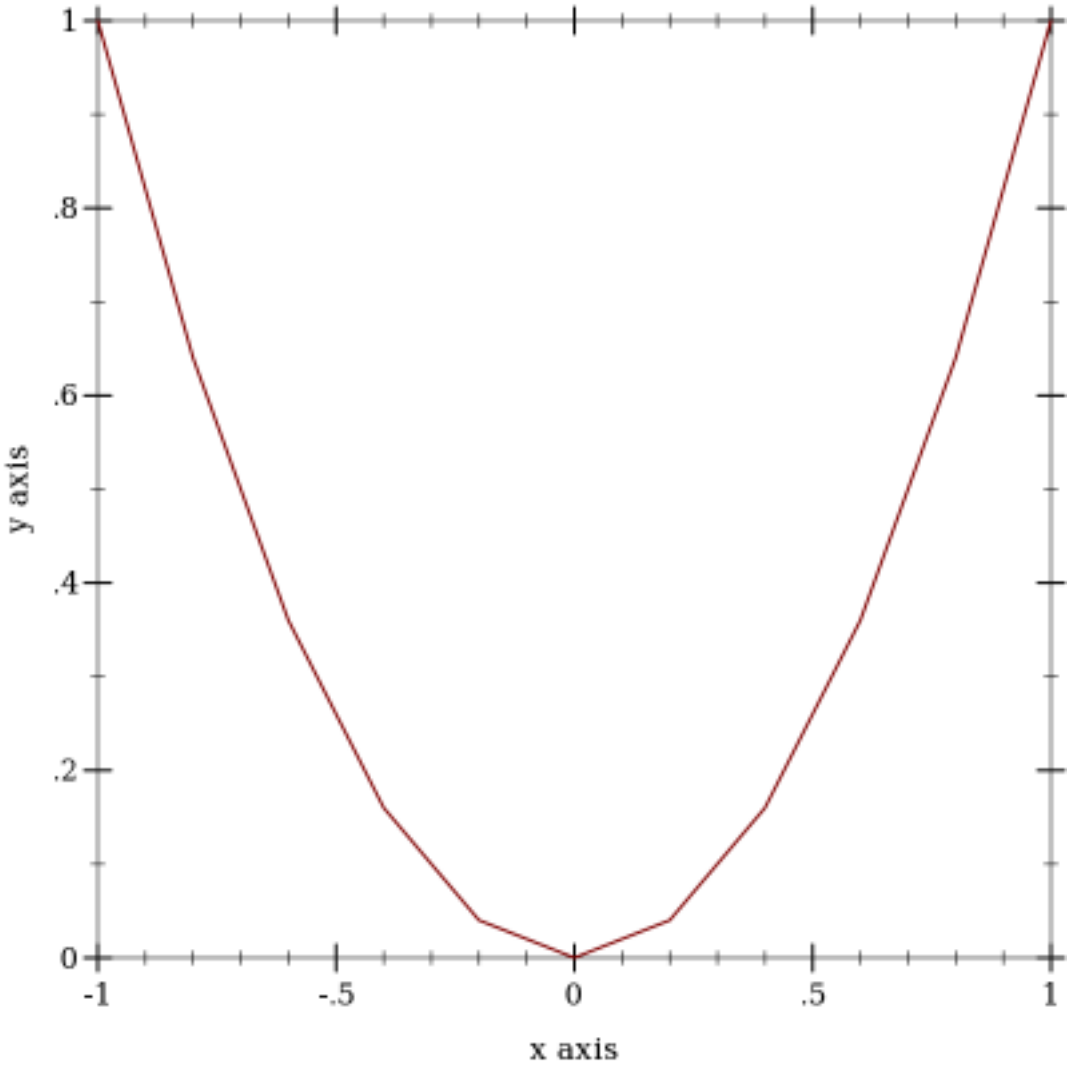
Returns a list of uniformly spaced real numbers between *start* and *end*. If *start?* is `#t`, the list includes *start*. If *end?* is `#t`, the list includes *end*.

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)
> (define xs (linear-seq -1 1 11))

> (plot (lines (map vector xs (map sqr xs))))
```



```
(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:

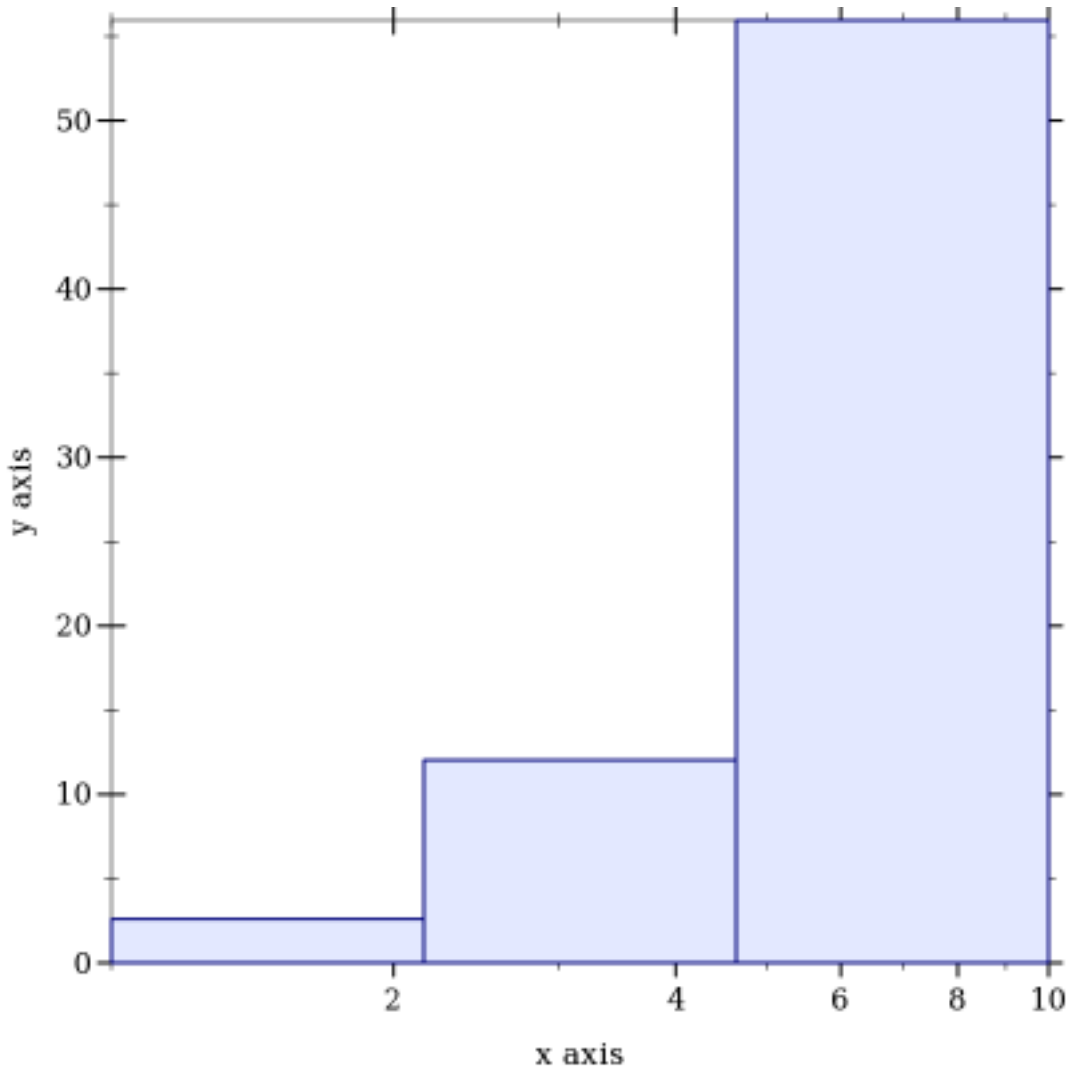
```
> (linear-seq* '(0 1 2) 5)
'(0 1/2 1 3/2 2)
> (linear-seq* '(0 1 2) 6)
'(0 2/5 4/5 6/5 8/5 2)
> (linear-seq* '(0 1 0) 5)
'(0 1/2 1 1/2 0)
```

```
(nonlinear-seq start
               end
               num
               transform
              [#:start? start?
               #:end? end?])    → (listof real?)
  start : real?
  end : real?
  num : exact-nonnegative-integer?
  transform : axis-transform/c
  start? : boolean? = #t
  end? : boolean? = #t
```

Generates a list of reals that, if transformed using *transform*, would be uniformly 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 rational? #f)
                (or/c rational? #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`.

8.3 Plot Colors and Styles

```
(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:

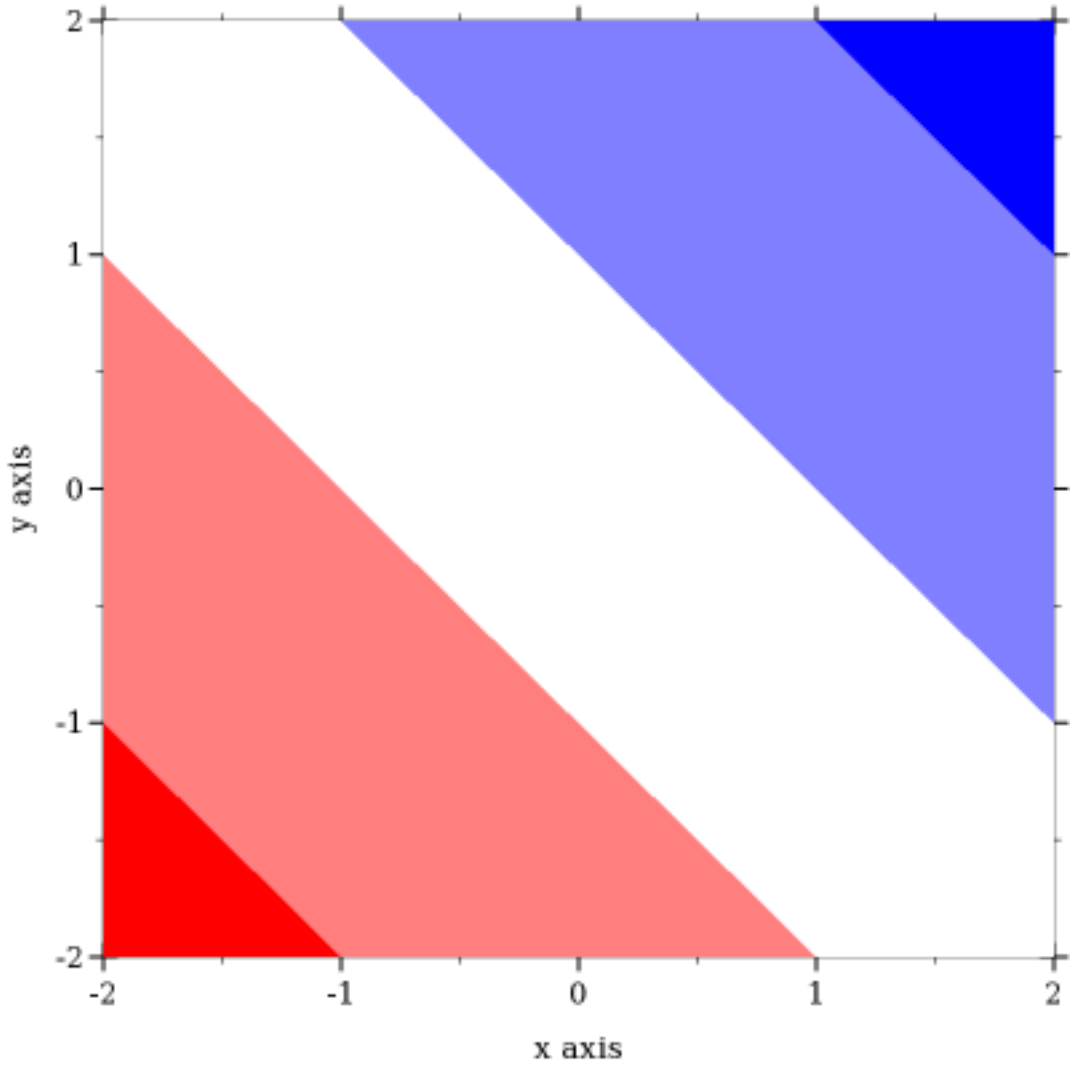
```
> (plot (contour-intervals (λ (x y) (+ x y)) -2 2 -2 2
                           #:levels 4 #:contour-
styles '(transparent)
                           #:colors (color-seq "red" "blue" 5)))
/: division by zero
```

```
(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.

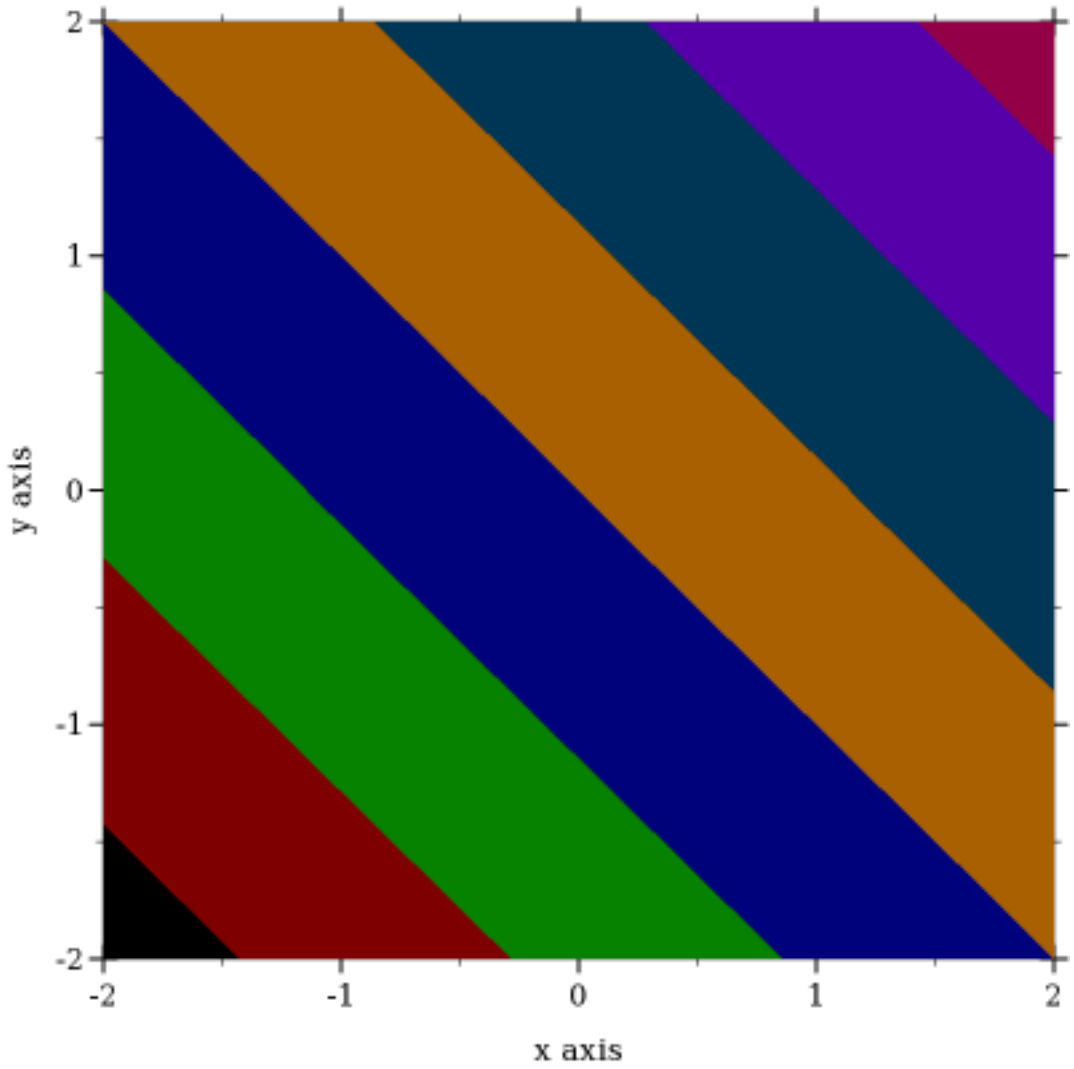
```
(->pen-color c) → (list/c real? real? real?)
  c : plot-color/c
```

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 `128`.

Examples:

```
> (equal? (->pen-color 0) (->pen-color 8))
#f
> (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
```

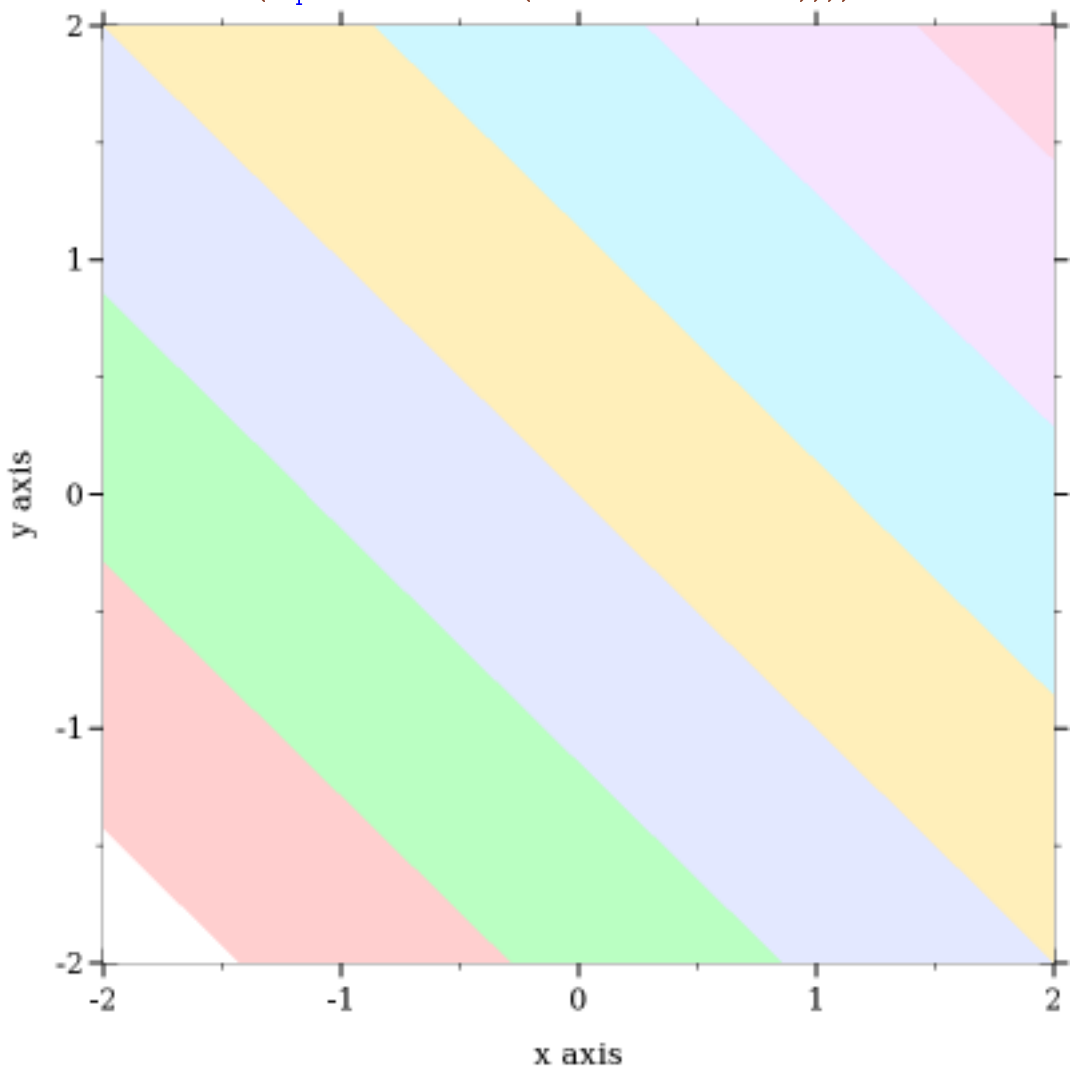
Converts a *fill* color to an RGB triplet. This function interprets integer colors as lighter and less saturated than `->pen-color` does.

Non-integer colors are converted using `->color`. Integer colors are chosen for good pairwise contrast, especially between neighbors. Integer colors repeat starting with `128`.

Examples:

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

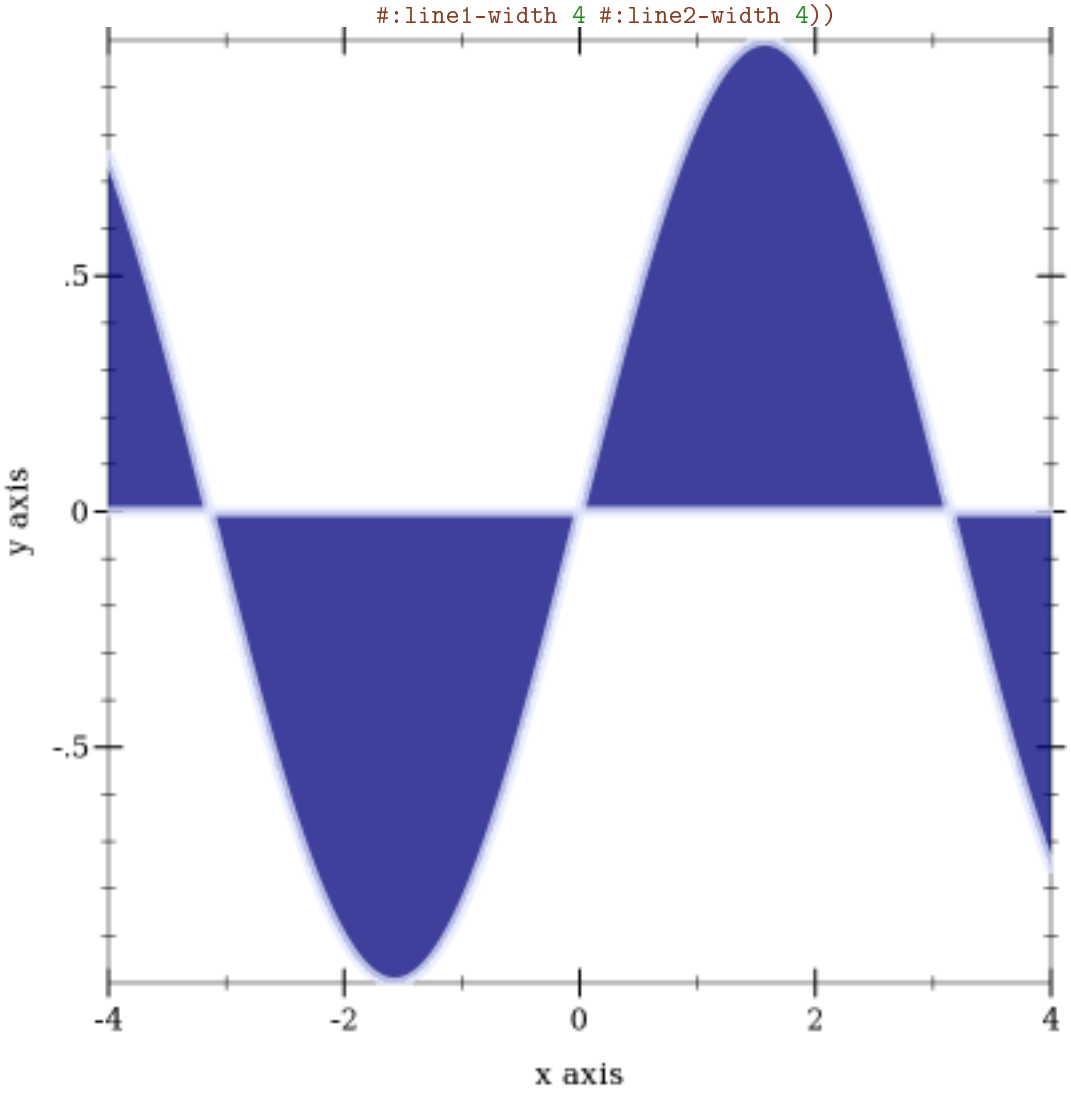
```
> (plot (contour-intervals
         (λ (x y) (+ x y)) -2 2 -2 2
         #:levels 7 #:contour-styles '(transparent)
         #:colors (map ->brush-color (build-list 8 values))))
```



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)
```

8.4 Plot-Specific Math

8.4.1 Real Functions

```
(polar->cartesian θ r) → (vector/c real? real?)
  θ : real?
  r : real?
```

Converts 2D polar coordinates to 3D cartesian coordinates.

```
(3d-polar->3d-cartesian θ ρ r) → (vector/c real? real? real?)
  θ : real?
  ρ : real?
  r : real?
```

Converts 3D polar coordinates to 3D cartesian coordinates. See parametric3d for an example of use.

```
(ceiling-log/base b x) → exact-integer?
  b : (and/c exact-integer? (>=/c 2))
  x : (>/c 0)
```

Like `(ceiling (/ (log x) (log b)))`, but `ceiling-log/base` is not susceptible to floating-point error.

Examples:

```
> (ceiling (/ (log 100) (log 10)))
2.0
> (ceiling-log/base 10 100)
2
> (ceiling (/ (log 1/1000) (log 10)))
-2.0
> (ceiling-log/base 10 1/1000)
-3
```

Various number-formatting functions use this.

```
(floor-log/base b x) → exact-integer?
  b : (and/c exact-integer? (>=/c 2))
  x : (>/c 0)
```

Like `(floor (/ (log x) (log b)))`, but `floor-log/base` is not susceptible to floating-point error.

Examples:

```
> (floor (/ (log 100) (log 10)))
2.0
> (floor-log/base 10 100)
2
> (floor (/ (log 1000) (log 10)))
2.0
> (floor-log/base 10 1000)
3
```

This is a generalization of `order-of-magnitude`.

```
(maybe-inexact->exact x) → (or/c rational? #f)
  x : (or/c rational? #f)
```

Returns `#f` if `x` is `#f`; otherwise `(inexact->exact x)`. Use this to convert interval endpoints, which may be `#f`, to exact numbers.

8.4.2 Vector Functions

```
(v+ v1 v2) → (vectorof real?)
  v1 : (vectorof real?)
  v2 : (vectorof real?)
```

```
(v- v1 v2) → (vectorof real?)
  v1 : (vectorof real?)
  v2 : (vectorof real?)
```

```
(vneg v) → (vectorof real?)
  v : (vectorof real?)
```

```
(v* v c) → (vectorof real?)
  v : (vectorof real?)
  c : real?
```

```
(v/ v c) → (vectorof real?)
  v : (vectorof real?)
  c : real?
```

Vector arithmetic. Equivalent to `vector-mapp`-ing arithmetic operators over vectors, but specialized so that 2- and 3-vector operations are much faster.

Examples:

```
> (v+ #(1 2) #(3 4))
'#(4 6)
> (v- #(1 2) #(3 4))
'#(-2 -2)
> (vneg #(1 2))
'#(-1 -2)
> (v* #(1 2 3) 2)
'#(2 4 6)
> (v/ #(1 2 3) 2)
'#(1/2 1 3/2)
```

```
(v= v1 v2) → boolean?
  v1 : (vectorof real?)
  v2 : (vectorof real?)
```

Like `equal?` specialized to numeric vectors, but compares elements using `=`.

Examples:

```
> (equal? #(1 2) #(1 2))
#t
```

```
> (equal? #(1 2) #(1.0 2.0))
#f
> (v= #(1 2) #(1.0 2.0))
#t
```

```
(vcross v1 v2) → (vector/c real? real? real?)
  v1 : (vector/c real? real? real?)
  v2 : (vector/c real? real? real?)
```

Returns the right-hand vector cross product of *v1* and *v2*.

Examples:

```
> (vcross #(1 0 0) #(0 1 0))
'#(0 0 1)
> (vcross #(0 1 0) #(1 0 0))
'#(0 0 -1)
> (vcross #(0 0 1) #(0 0 1))
'#(0 0 0)
```

```
(vcross2 v1 v2) → real?
  v1 : (vector/c real? real?)
  v2 : (vector/c real? real?)
```

Returns the signed area of the 2D parallelogram with sides *v1* and *v2*. Equivalent to `(vector-ref (vcross (vector-append v1 #(0)) (vector-append v2 #(0))) 2)` but faster.

Examples:

```
> (vcross2 #(1 0) #(0 1))
1
> (vcross2 #(0 1) #(1 0))
-1
```

```
(vdot v1 v2) → real?
  v1 : (vectorof real?)
  v2 : (vectorof real?)
```

Returns the dot product of *v1* and *v2*.

```
(vmag^2 v) → real?
  v : (vectorof real?)
```

Returns the squared magnitude of *v*. Equivalent to `(vdot v v)`.

```
(vmag v) → real?
  v : (vectorof real?)
```

Returns the magnitude of `v`. Equivalent to `(sqrt (vmag^2 v))`.

```
(vnormalize v) → (vectorof real?)
  v : (vectorof real?)
```

Returns a normal vector in the same direction as `v`. If `v` is a zero vector, returns `v`.

Examples:

```
> (vnormalize #(1 1 0))
'#(0.7071067811865475 0.7071067811865475 0)
> (vnormalize #(1 1 1))
'#(0.5773502691896258 0.5773502691896258 0.5773502691896258)
> (vnormalize #(0 0 0.0))
'#(0 0 0.0)
```

```
(vcenter vs) → (vectorof real?)
  vs : (listof (vectorof real?))
```

Returns the center of the smallest bounding box that contains `vs`.

Example:

```
> (vcenter '(#(1 1) #(2 2)))
'#(3/2 3/2)
```

```
(vrational? v) → boolean?
  v : (vectorof real?)
```

Returns `#t` if every element of `v` is `rational?`.

Examples:

```
> (vrational? #(1 2))
#t
> (vrational? #(+inf.0 2))
#f
> (vrational? #(#f 1))
#f
```

8.4.3 Intervals and Interval Functions

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

Represents a closed interval.

An interval with two real-valued endpoints always contains the endpoints in order:

```
> (ivl 0 1)
(ivl 0 1)
> (ivl 1 0)
(ivl 0 1)
```

An interval can have infinite endpoints:

```
> (ivl -inf.0 0)
(ivl -inf.0 0)
> (ivl 0 +inf.0)
(ivl 0 +inf.0)
> (ivl -inf.0 +inf.0)
(ivl -inf.0 +inf.0)
```

Functions that return rectangle renderers, such as `rectangles` and `discrete-histogram3d`, accept vectors of `ivl`s as arguments. The `ivl` struct type is also provided by `plot` so users of such renderers do not have to require `plot/utils`.

```
(rational-ivl? i) → boolean?
  i : any/c
```

Returns `#t` if *i* is an interval and each of its endpoints is `rational?`.

Example:

```
> (map rational-ivl? (list (ivl -1 1) (ivl -inf.0 2) 'bob))
'(#t #f #f)
```

```
(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))
```

```
(clamp-real x i) → real?
  x : real?
  i : ivl?
```

8.5 Dates and Times

```
(datetime->real x) → real?
  x : (or/c plot-time? date? date*? sql-date? sql-time? sql-timestamp?)
```

Converts various date/time representations into UTC seconds, respecting time zone offsets.

For dates, the value returned is the number of seconds since *a system-dependent UTC epoch*. See `date-ticks` for more information.

To plot a time series using dates pulled from an SQL database, simply set the relevant axis ticks (probably `plot-x-ticks`) to `date-ticks`, and convert the dates to seconds using `datetime->real` before passing them to `lines`. To keep time zone offsets from influencing the plot, set them to `0` first.

```
(struct plot-time (second minute hour day)
        #:extra-constructor-name make-plot-time)
  second : (and/c (>=/c 0) (</c 60))
  minute : (integer-in 0 59)
  hour : (integer-in 0 23)
  day : exact-integer?
```

A time representation that accounts for days, negative times (using negative days), and fractional seconds.

PLoT (specifically `time-ticks`) uses `plot-time` internally to format times, but because renderer-producing functions require only real values, user code should not need it. It is provided just in case.

```
(plot-time->seconds t) → real?
  t : plot-time?
```

```
(seconds->plot-time s) → plot-time?
  s : real?
```

Convert `plot-time`s to real seconds, and vice-versa.

Examples:

```
> (define (plot-time+ t1 t2)
    (seconds->plot-time (+ (plot-time->seconds t1)
                           (plot-time->seconds t2))))

> (plot-time+ (plot-time 32 0 12 1)
              (plot-time 32 0 14 1))
(plot-time 4 1 2 3)
```

9 Plot and Renderer Parameters

9.1 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.

9.2 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%`.

9.3 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-left
```

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

```
= #f
```

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

```
= #f
```

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

```
= #f
```

The axis labels for “far” axes. See plot-x-ticks for a discussion of near and far axes.

```
(plot-x-tick-label-anchor) → anchor/c
(plot-x-tick-label-anchor anchor) → void?
  anchor : anchor/c
```

```
= 'top
```

```
(plot-x-tick-label-angle) → real?
(plot-x-tick-label-angle angle) → void?
  angle : real?
```

```
= 0
```

```
(plot-y-tick-label-anchor) → anchor/c
(plot-y-tick-label-anchor anchor) → void?
  anchor : anchor/c
```

```
= 'right
```

```
(plot-y-tick-label-angle) → real?
(plot-y-tick-label-angle angle) → void?
  angle : real?
```

```
= 0
```

```
(plot-x-far-tick-label-anchor) → anchor/c
(plot-x-far-tick-label-anchor anchor) → void?
  anchor : anchor/c

=  'bottom
```

```
(plot-x-far-tick-label-angle) → real?
(plot-x-far-tick-label-angle angle) → void?
  angle : real?

=  0
```

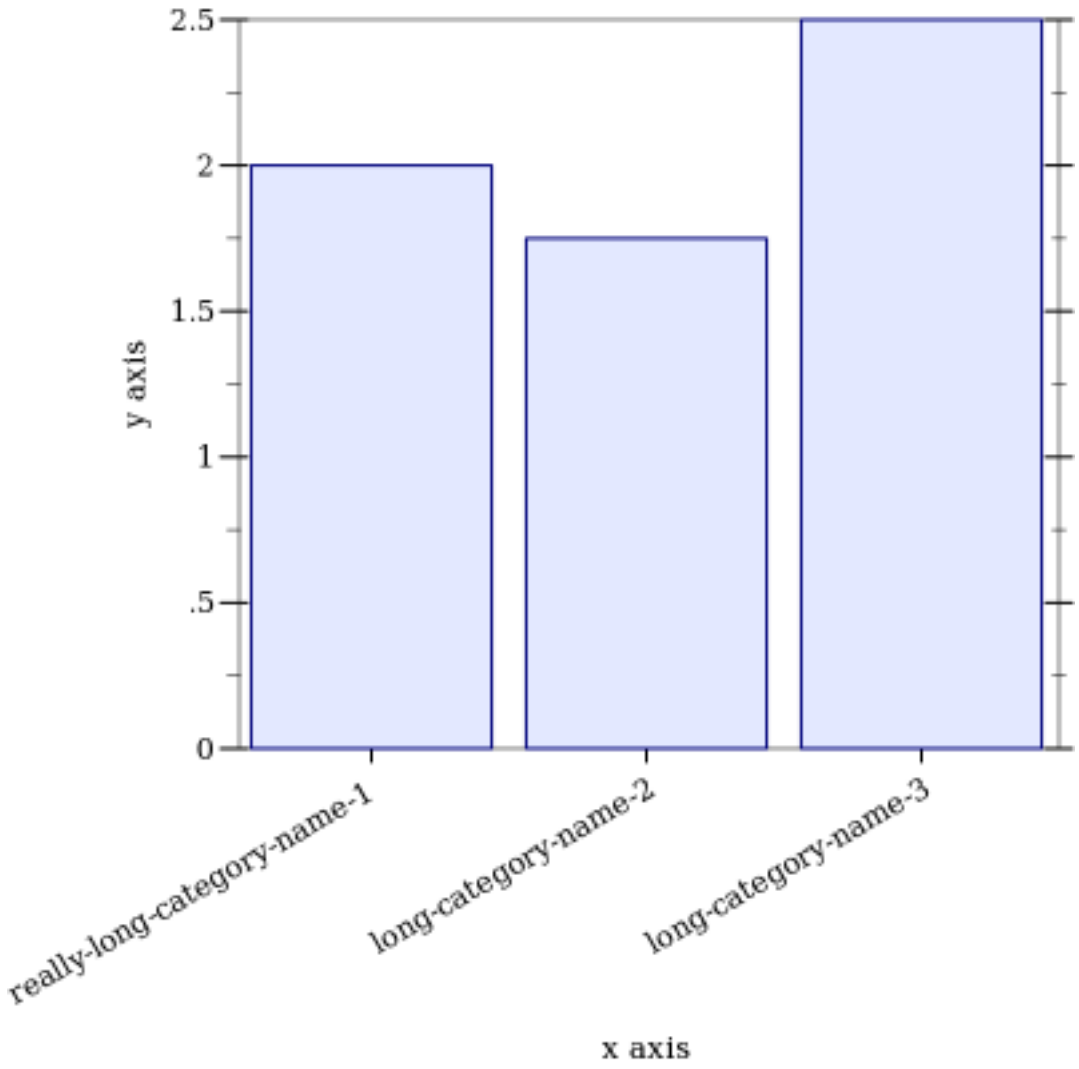
```
(plot-y-far-tick-label-anchor) → anchor/c
(plot-y-far-tick-label-anchor anchor) → void?
  anchor : anchor/c

=  'left
```

```
(plot-y-far-tick-label-angle) → real?
(plot-y-far-tick-label-angle angle) → void?
  angle : real?

=  0
```

Anchor and angles for axis tick labels (2D only). Angles are in degrees. The anchor refers to the part of the label attached to the end of the tick line.

Set these when labels would otherwise overlap; for example, in histograms with long category names.

```
> (parameterize ([plot-x-tick-label-anchor  'top-right]
                 [plot-x-tick-label-angle   30])
    (plot (discrete-histogram '(#(really-long-category-name-1 2)
                                #(long-category-name-2 1.75)
                                #(long-category-name-3 2.5)))))
```




```
(plot-x-axis?) → boolean?
(plot-x-axis? axis?) → void?
  axis? : boolean?

= #t
```

```
(plot-x-far-axis?) → boolean?
(plot-x-far-axis? axis?) → void?
  axis? : boolean?
```

```
= #t
```

```
(plot-y-axis?) → boolean?
(plot-y-axis? axis?) → void?
  axis? : boolean?

= #t
```

```
(plot-y-far-axis?) → boolean?
(plot-y-far-axis? axis?) → void?
  axis? : boolean?

= #t
```

```
(plot-z-axis?) → boolean?
(plot-z-axis? axis?) → void?
  axis? : boolean?

= #t
```

```
(plot-z-far-axis?) → boolean?
(plot-z-far-axis? axis?) → void?
  axis? : boolean?

= #t
```

When any of these is `#f`, the corresponding axis is not drawn.

Use these along with `x-axis` and `y-axis` if you want axes that intersect the origin or some other point.

```
(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.

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

```
= (cond [(plot-animating?) (max 2 (ceiling (* 1/4 samples)))]
        [else samples])
```

Given a number of samples, returns the number of samples to use. This returns `samples` when `plot-animating?` is `#f`.

```
(plot-decorations?) → boolean?
(plot-decorations? decorations?) → void?
  decorations? : boolean?

= #t
```

When `#f`, axes, axis labels, ticks, tick labels, and the title are not drawn.

9.4 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
```

9.5 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
```

9.6 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
```

9.7 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
```

```
(vector-field3d-samples) → exact-positive-integer?
(vector-field3d-samples samples) → void?
  samples : exact-positive-integer?
= 9
```

9.8 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
```

9.9 Contours and Contour Intervals

```
(default-contour-colors zs) → (listof plot-color/c)
  zs : (listof real?)
= (color-seq* (list (->pen-color 5) (->pen-color 0) (->pen-color 1))
              (length zs))
```

```
(default-contour-fill-colors z-ivls) → (listof plot-color/c)
  z-ivls : (listof ivl?)
= (color-seq* (list (->brush-color 5) (->brush-color 0) (->brush-color 1))
              (length z-ivls))
```

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

=  default-contour-colors
```

```
(contour-widths) → (pen-widths/c (listof real?))
(contour-widths widths) → void?
  widths : (pen-widths/c (listof real?))

=  '(1)
```

```
(contour-styles) → (plot-pen-styles/c (listof real?))
(contour-styles styles) → void?
  styles : (plot-pen-styles/c (listof real?))

=  '(solid long-dash)
```

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

=  '(1)
```

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

=  default-contour-fill-colors
```

```
(contour-interval-styles)
 → (plot-brush-styles/c (listof ivl?))
(contour-interval-styles styles) → void?
  styles : (plot-brush-styles/c (listof ivl?))

= '(solid)
```

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

= '(1)
```

9.10 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
```

```
(discrete-histogram-skip) → (>=/c 0)
(discrete-histogram-skip skip) → void?
  skip : (>=/c 0)
= 1
```

```
(discrete-histogram-invert?) → boolean?
(discrete-histogram-invert? invert?) → void?
  invert? : boolean?
= #f
```

```
(stacked-histogram-alphas) → (alphas/c nat/c)
(stacked-histogram-alphas alphas) → void?
  alphas : (alphas/c nat/c)
```

```
= '(1)
```

```
(stacked-histogram-colors) → (plot-colors/c nat/c)
(stacked-histogram-colors colors) → void?
  colors : (plot-colors/c nat/c)

= (λ (n) (build-list n add1))
```

```
(stacked-histogram-line-colors) → (plot-colors/c nat/c)
(stacked-histogram-line-colors colors) → void?
  colors : (plot-colors/c nat/c)

= (stacked-histogram-colors)
```

```
(stacked-histogram-line-styles) → (plot-pen-styles/c nat/c)
(stacked-histogram-line-styles styles) → void?
  styles : (plot-pen-styles/c nat/c)

= '(solid)
```

```
(stacked-histogram-line-widths) → (pen-widths/c nat/c)
(stacked-histogram-line-widths widths) → void?
  widths : (pen-widths/c nat/c)

= '(1)
```

```
(stacked-histogram-styles) → (plot-brush-styles/c nat/c)
(stacked-histogram-styles styles) → void?
  styles : (plot-brush-styles/c nat/c)

= '(solid)
```

9.11 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-alpha) → (real-in 0 1)
(x-axis-alpha alpha) → void?
  alpha : (real-in 0 1)

= 1
```

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

= 1
```

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

= 1
```

```
(x-axis-far?) → boolean?
(x-axis-far? far?) → void?
  far? : boolean?

= #f
```

```
(y-axis-far?) → boolean?
(y-axis-far? far?) → void?
  far? : boolean?

= #f
```

```
(z-axis-far?) → boolean?
(z-axis-far? far?) → void?
  far? : boolean?

= #f
```

```
(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
```

```
(x-axis-labels?) → boolean?
(x-axis-labels? labels?) → void?
  labels? : boolean?

= #f
```

```
(y-axis-labels?) → boolean?
(y-axis-labels? labels?) → void?
  labels? : boolean?

= #f
```

```
(z-axis-labels?) → boolean?
(z-axis-labels? labels?) → void?
  labels? : boolean?

= #f
```

```
(polar-axes-number) → exact-nonnegative-integer?
(polar-axes-number number) → void?
  number : exact-nonnegative-integer?

= 12
```

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

= 1/2
```

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

= #t
```

```
(polar-axes-labels?) → boolean?
(polar-axes-labels? labels?) → void?
  labels? : 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
```

9.12 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
```

9.13 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
```

9.14 Contour Surfaces

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

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

= '(0)
```

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

= '(1/3)
```

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

= '(solid)
```

9.15 Isosurfaces

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

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

= (color-seq* (list (->brush-color 5) (->brush-color 0) (->brush-color 1))
              (length zs))
```

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

=  (color-seq* (list (->pen-color 5) (->pen-color 0) (->pen-color 1))
               (length zs))
```

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

=  'auto
```

```
(isosurface-colors) → (plot-colors/c (listof real?))
(isosurface-colors colors) → void?
  colors : (plot-colors/c (listof real?))

=  default-isosurface-colors
```

```
(isosurface-styles) → (plot-brush-styles/c (listof real?))
(isosurface-styles styles) → void?
  styles : (plot-brush-styles/c (listof real?))

=  '(solid)
```

```
(isosurface-line-colors) → (plot-colors/c (listof real?))
(isosurface-line-colors colors) → void?
  colors : (plot-colors/c (listof real?))

=  default-isosurface-line-colors
```

```
(isosurface-line-widths) → (pen-widths/c (listof real?))
(isosurface-line-widths widths) → void?
  widths : (pen-widths/c (listof real?))

=  '(1/3)
```

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

=  '(solid)
```

```
(isosurface-alphas) → (alphas/c (listof real?))
(isosurface-alphas alphas) → void?
  alphas : (alphas/c (listof real?))

=  '(1/2)
```

10 Plot Contracts

10.1 Plot Element Contracts

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

Returns `#t` if *value* is a 2D renderer; that is, if `plot` can plot *value*. See §3 “2D Renderers” for functions that construct them.

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

Returns `#t` if *value* is a 3D renderer; that is, if `plot3d` can plot *value*. See §5 “3D Renderers” for functions that construct them.

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

Returns `#t` if *value* is a nonrenderer. See §6 “Nonrenderers” for functions that construct them.

10.2 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%))
```

A contract for very flexible color arguments. Functions that accept a `color/c` almost always convert it to an RGB triplet using `->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?)
```

```
= (list '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)
```

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

10.3 Appearance Argument List Contracts

```
(maybe-function/c in-contract out-contract) → contract?
  in-contract : contract?
  out-contract : contract?

= (or/c out-contract (in-contract . -> . out-contract))
```

Returns a contract that accepts either a function from `in-contract` to `out-contract`, or a plain `out-contract` value.

```
> (require racket/contract)

> (define/contract (maybe-function-of-real-consumer x)
      ((maybe-function/c real? real?) . -> . real?)
    (maybe-apply x 10))

> (maybe-function-of-real-consumer 4)
4
> (maybe-function-of-real-consumer (λ (x) x))
```

```
10
```

Many plot functions, such as contours and isosurfaces3d, optionally take lists of appearance values (such as (listof plot-color/c)) as arguments. A very flexible argument contract would accept *functions* that produce lists of appearance values. For example, contours would accept any f with contract (-> (listof real?) (listof plot-color/c)) for its #:colors argument. When rendering a contour plot, contours would apply f to a list of the contour z values to get the contour colors.

However, most uses do not need this flexibility. Therefore, plot's functions accept *either* a list of appearance values *or* a function from a list of appropriate values to a list of appearance values. The maybe-function/c function constructs contracts for such arguments.

In plot functions, if in-contract is a listof contract, the output list's length need not be the same as the input list's length. If it is shorter, the appearance values will cycle; if longer, the tail will not be used.

```
(maybe-apply f arg) → any/c
  f : (maybe-function/c any/c any/c)
  arg : any/c
```

If f is a function, applies f to args; otherwise returns f.

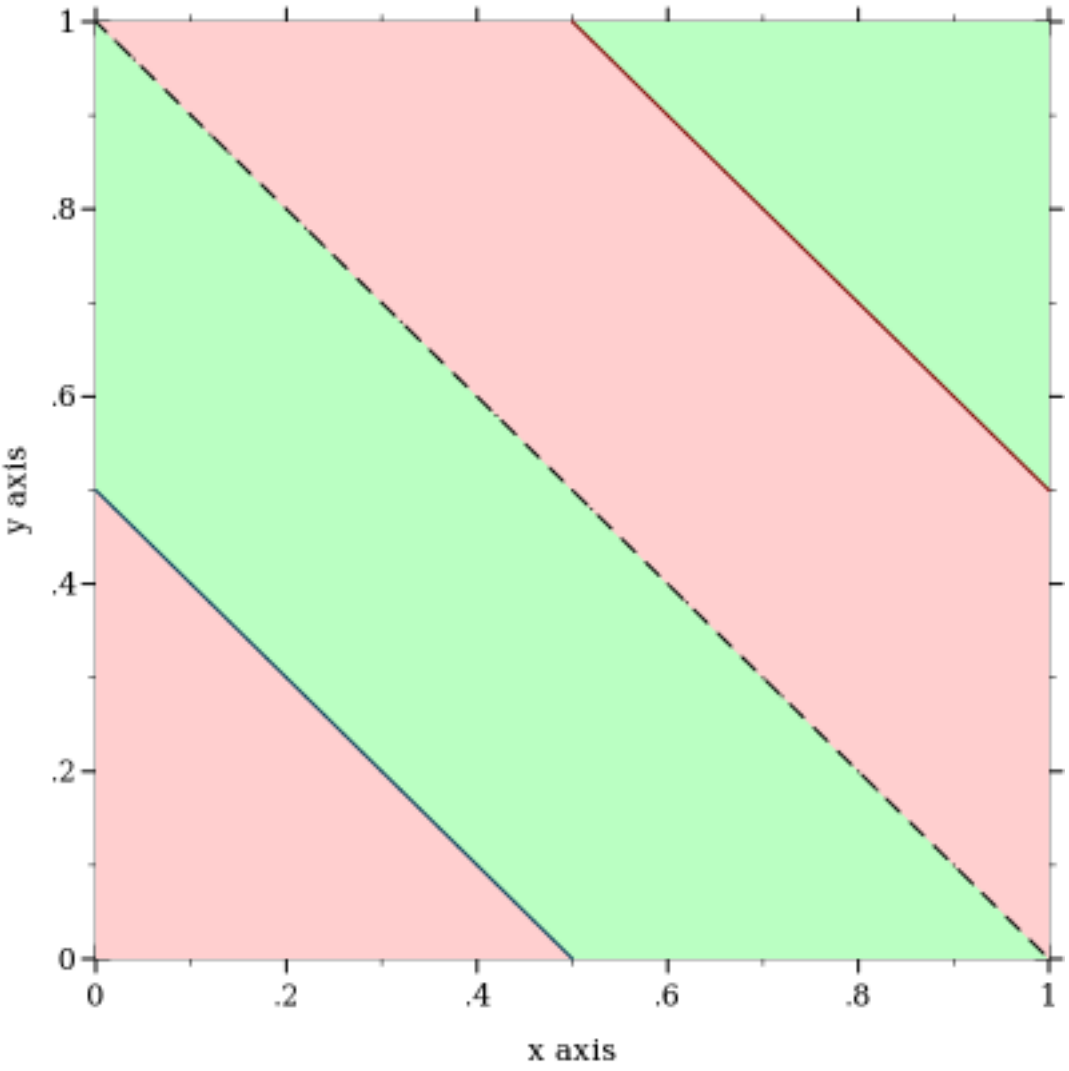
This is used inside many renderer-producing plot functions to convert maybe-function/c values to lists of appearance values.

```
(plot-colors/c in-contract) → contract?
  in-contract : contract?

= (maybe-function/c in-contract (listof plot-color/c))
```

Returns a contract for #:colors arguments, as in contours and contour-intervals. See maybe-function/c for a discussion of the returned contract.

The following example sends a *list*-valued (plot-colors/c ivl?) to contour-intervals, which then cycles through the colors:

```
> (plot (contour-intervals (λ (x y) (+ x y)) 0 1 0 1
                           #:colors '(1 2)))
```

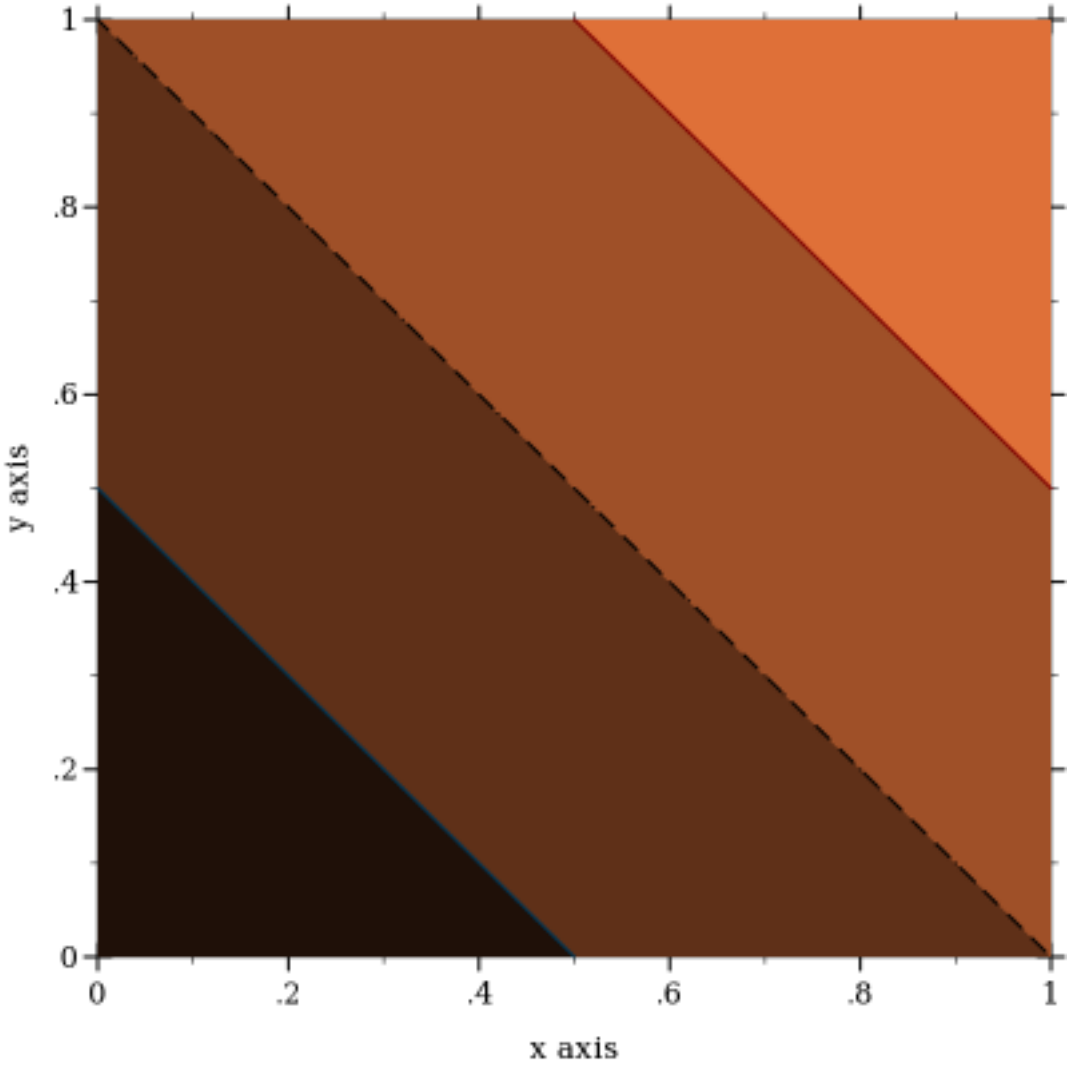


This is equivalent to sending `(λ _ '(1 2))`.

The next example is more sophisticated: it sends a *function*-valued `(plot-colors/civl?)` to `contour-intervals`. The function constructs colors from the values of the contour intervals.

```
> (define (brown-interval-colors ivls)
    (define z-size (- (ivl-max (last ivls))
                      (ivl-min (first ivls))))
    (for/list ([i  (in-list ivls)])
      (match-define (ivl z-min z-max) i)
      (define z-mid (/ (* 1/2 (+ z-min z-max)) z-size))
```

```
      (list (* 255 z-mid) (* 128 z-mid) (* 64 z-mid))))

> (plot (contour-intervals (λ (x y) (+ x y)) 0 1 0 1
                           #:colors brown-interval-colors))
```




```
(plot-pen-styles/c in-contract) → contract?
  in-contract : contract?

= (maybe-function/c in-contract (listof plot-pen-style/c))
```

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

```
(pen-widths/c in-contract) → contract?
  in-contract : contract?

= (maybe-function/c in-contract (listof (>=/c 0)))
```

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

```
(plot-brush-styles/c in-contract) → contract?
  in-contract : contract?

= (maybe-function/c in-contract (listof plot-brush-style/c))
```

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

```
(alphas/c in-contract) → contract?
  in-contract : contract?

= (maybe-function/c in-contract (listof (real-in 0 1)))
```

Like plot-colors/c, but for opacities.

```
(labels/c in-contract) → contract?
  in-contract : contract?

= (maybe-function/c in-contract (listof (or/c string? #f)))
```

Like plot-colors/c, but for strings. This is used, for example, to label stacked-histograms.

11 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 §12 “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.
- The `fit` function and `fit-result` struct type have been removed.

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.

11.1 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.

11.2 Ensuring That Plots Have Bounds

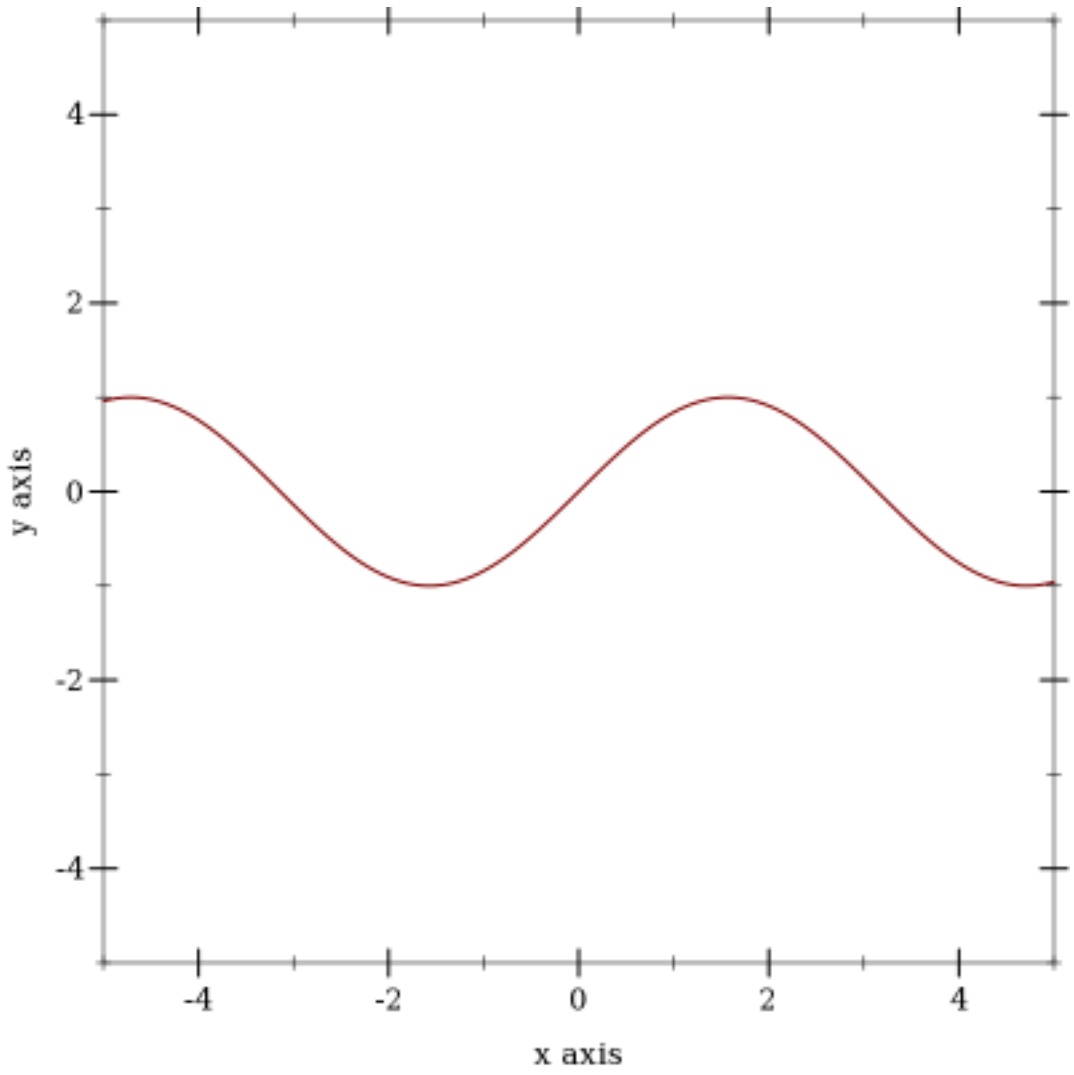
The safest way to ensure that `plot` can determine bounds for the plot area is to add `#:x-min -5 #:x-max 5 #:y-min -5 #:y-max 5` to every call to `plot`. Similarly, add `#:x-min -5 #:x-max 5 #:y-min -5 #:y-max 5 #:z-min -5 #:z-max 5` to every call to `plot3d`.

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

```
> (plot (line sin))
plot: could not determine sensible plot bounds; got x ∈
[#f,#f], y ∈ [#f,#f]
```

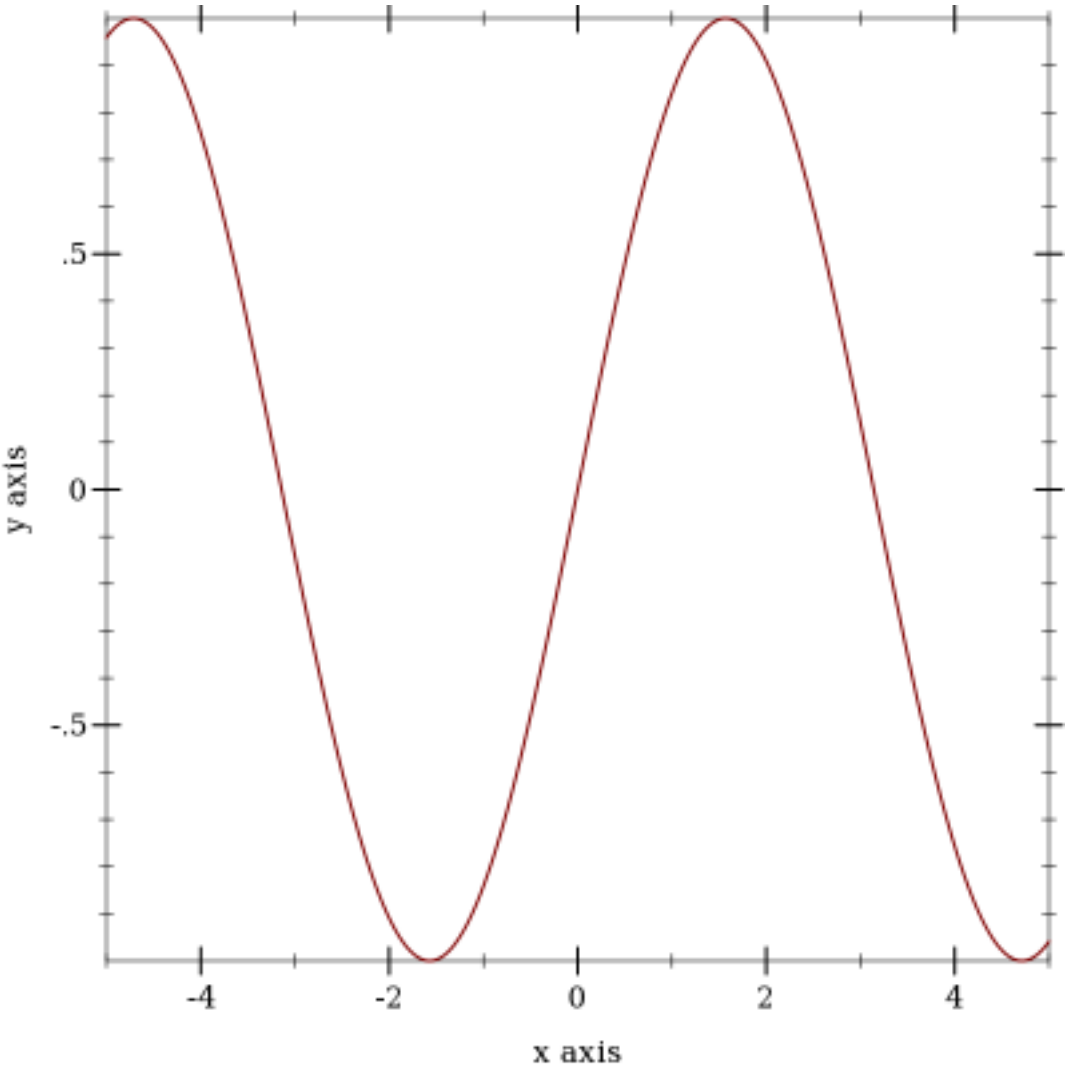
You could either change it to

```
> (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.

11.3 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).

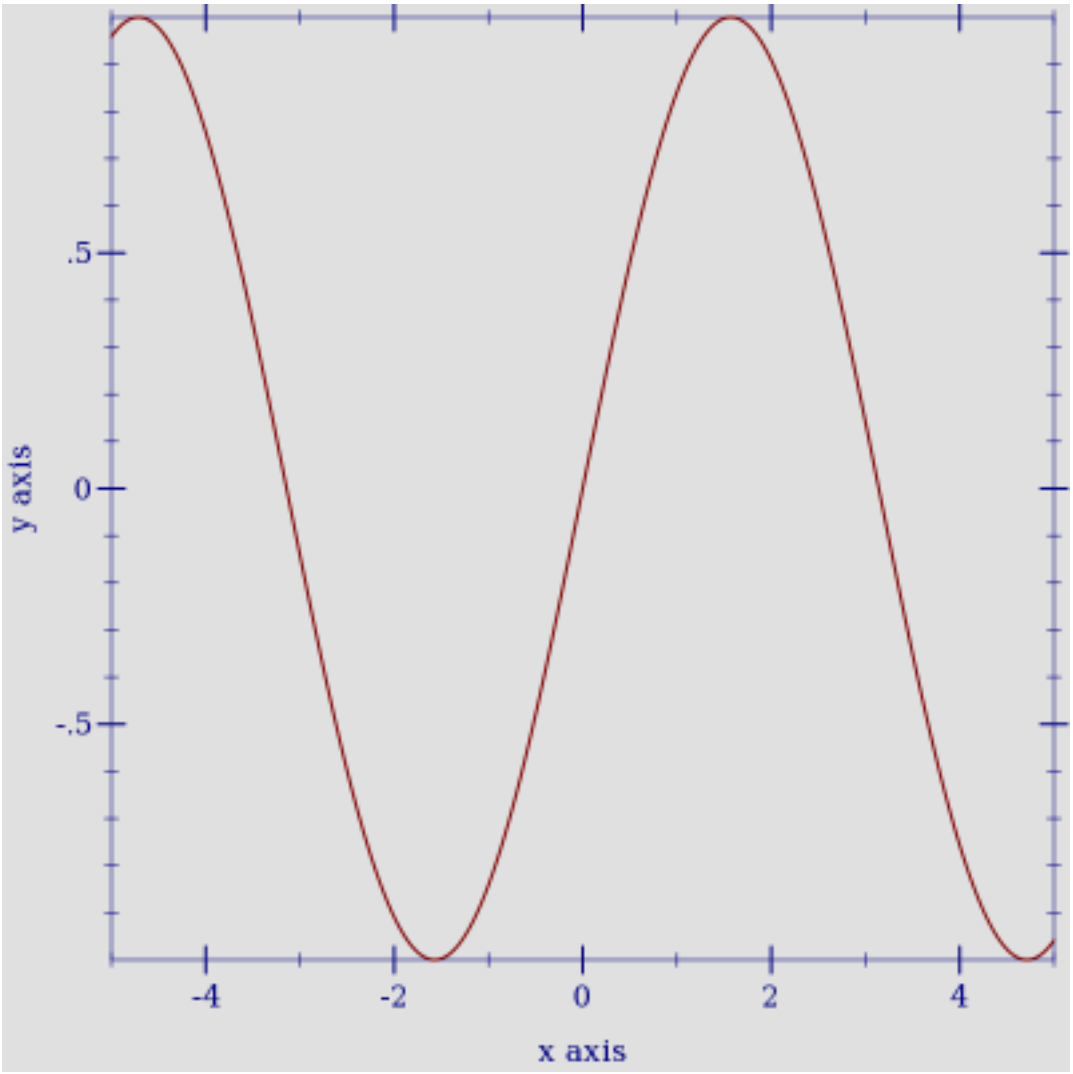
Replace every `#:style 'real` with `#:scale 1.0`.

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.

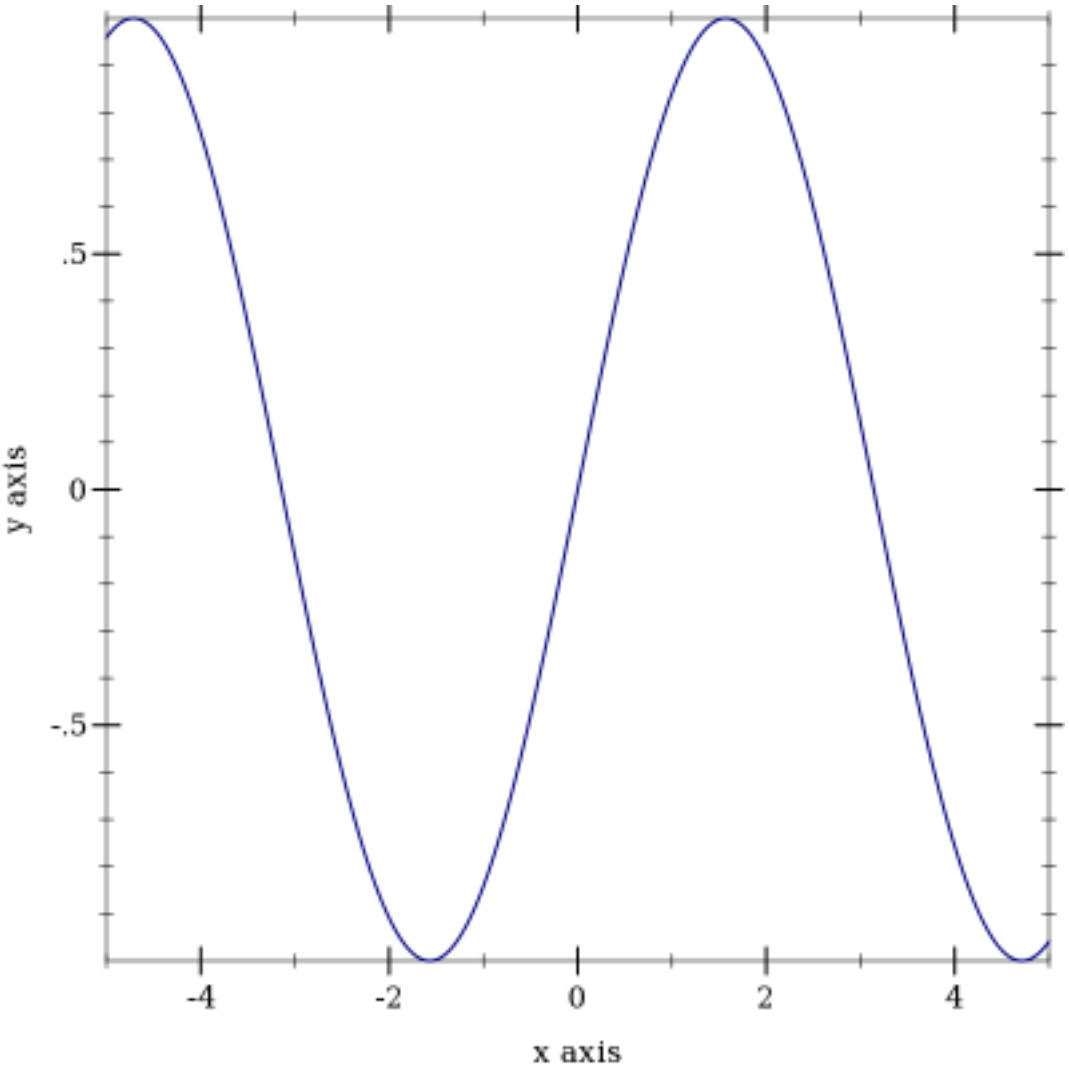
For example, if you have `(plot (function sin -5 5) #:fgcolor '(0 0 128) #:bgcolor '(224 224 224))`, change it to

```
> (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.

11.4 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`.

11.5 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.

11.6 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 §12 “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 §12 “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 §12 “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 §12 “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 §12 “Compatibility Module” for the original documentation. Replace this with surface3d.

12 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.

12.1 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
```

12.2 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 computes 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-valued function from two parts.