# Plot: Graph Plotting

Version 9.4.0.4

Neil Toronto <neil.toronto@gmail.com>

November 3, 2026

```
(require plot)
```
package: `plot-gui-lib`

The Plot library provides a flexible interface for producing nearly any kind of plot. It includes many common kinds of plots 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 interactive snips in DrRacket, as picts in slideshows, as PNG, PDF, PS and SVG files, or on any device context.

**Plot is a Typed Racket library**, but it can be used in untyped Racket programs with **little to no performance loss**. The old typed interface module `plot/typed` is still available for old Typed Racket programs. New Typed Racket programs should use `plot`.

For plotting without a GUI, see `plot/no-gui`. For plotting in REPL-like environments outside of DrRacket, including Scribble manuals, see `plot/pict` and `plot/bitmap`.

# Contents


**1 Introduction** **6**

1.1 Plotting 2D Graphs . . . . . 6

1.2 Terminology . . . . . 7

1.3 Plotting 3D Graphs . . . . . 7

1.4 Plotting Multiple 2D Renderers . . . . . 9

1.5 Renderer and Plot Bounds . . . . . 12

1.6 Plotting Multiple 3D Renderers . . . . . 14

1.7 Plotting to Files . . . . . 15

1.8 Colors and Styles . . . . . 15

**2 2D and 3D Plotting Procedures** **18**

2.1 GUI Plotting Procedures . . . . . 18

2.2 Non-GUI Plotting Procedures . . . . . 23

2.3 Pict-Plotting Work-a-Likes . . . . . 25

2.4 Bitmap-Plotting Work-a-Likes . . . . . 26

**3 2D Renderers** **27**

3.1 2D Renderer Function Arguments . . . . . 27

3.2 2D Point Renderers . . . . . 28

3.3 2D Line Renderers . . . . . 36

3.4 2D Interval Renderers . . . . . 52

3.5 2D Contour (Isoline) Renderers . . . . . 62

3.6 2D Rectangle Renderers . . . . . 68

3.7 Violin and Box Plot Renderers . . . . . 78

3.8 2D Plot Decoration Renderers . . . . . 85

3.9 Interactive Overlays for 2D plots . . . . . 96

**4 3D Renderers** **98**

4.1 3D Renderer Function Arguments . . . . . 98

4.2 3D Point Renderers . . . . . 98

4.3 3D Line Renderers . . . . . 102

4.4 3D Surface Renderers . . . . . 106

4.5 3D Contour (Isoline) Renderers . . . . . 116

4.6 3D Isosurface Renderers . . . . . 122

4.7 3D Rectangle Renderers . . . . . 126

**5 Nonrenderers** **134**

**6 Axis Transforms and Ticks** **137**

6.1 Axis Transforms . . . . . 137

6.2 Axis Ticks . . . . . 150

6.2.1 Linear Ticks . . . . . 156

6.2.2 Log Ticks . . . . . 157

6.2.3 Date Ticks . . . . . 158

6.2.4 Time Ticks . . . . . 159

6.2.5 Currency Ticks . . . . . 161

6.2.6 Other Ticks . . . . . 163

6.2.7 Tick Combinators . . . . . 168

6.2.8 Tick Data Types and Contracts . . . . . 170

6.3 Invertible Functions . . . . . 170

**7 Plot Utilities** **172**

7.1 Formatting . . . . . 172
7.2 Sampling . . . . . 174
7.3 Plot Colors and Styles . . . . . 178
7.4 Plot-Specific Math . . . . . 185
7.4.1 Real Functions . . . . . 185
7.4.2 Vector Functions . . . . . 187
7.4.3 Intervals and Interval Functions . . . . . 190
7.5 Dates and Times . . . . . 191
7.6 Plot Metrics . . . . . 192

**8 Plot and Renderer Parameters** **196**

8.1 Compatibility . . . . . 196
8.2 Output . . . . . 196
8.3 General Appearance . . . . . 197
8.4 Lines . . . . . 208
8.5 Intervals . . . . . 209
8.6 Points and Point Labels . . . . . 210
8.7 Vector Fields & Arrows . . . . . 212
8.8 Error Bars . . . . . 213
8.9 Candlesticks . . . . . 214
8.10 Color fields . . . . . 215
8.11 Contours and Contour Intervals . . . . . 215
8.12 Contour Surfaces . . . . . 217
8.13 Rectangles . . . . . 217
8.14 Non-Border Axes . . . . . 219
8.15 Surfaces . . . . . 220

**9 Plot Contracts** **223**

9.1 Plot Element Contracts . . . . . 223

9.2 Appearance Argument Contracts . . . . . 223

9.3 Appearance Argument List Contracts . . . . . 228

**10 Porting From Plot <= 5.1.3** **232**

10.1 Replacing Deprecated Functions . . . . . 232

10.2 Ensuring That Plots Have Bounds . . . . . 233

10.3 Changing Keyword Arguments . . . . . 235

10.4 Fixing Broken Calls to `points` . . . . . 237

10.5 Replacing Uses of `plot-extend` . . . . . 238

10.6 Deprecated Functions . . . . . 238

**11 Legacy Typed Interface** **240**

**12 Compatibility Module** **241**

12.1 Plotting . . . . . 241

12.2 Miscellaneous Functions . . . . . 245

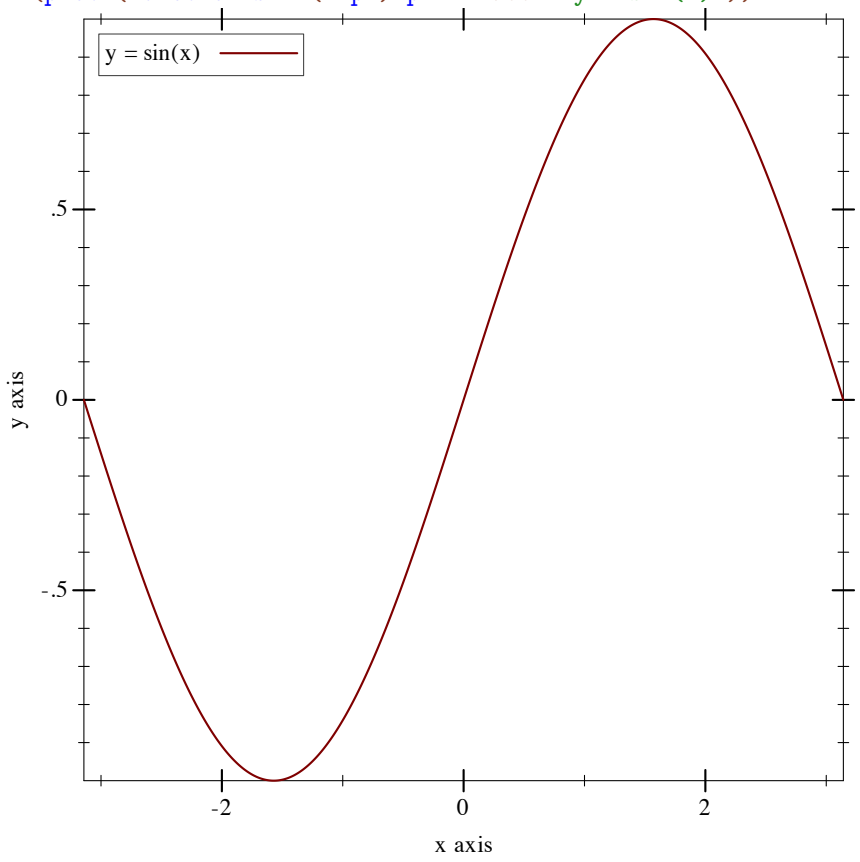
# 1 Introduction

```
(require plot)
```
package: `plot-gui-lib`

## 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)"))
```




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

If you're not using DrRacket, start with

```
(require plot)
```

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

to open each plot in a new window.

## 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.

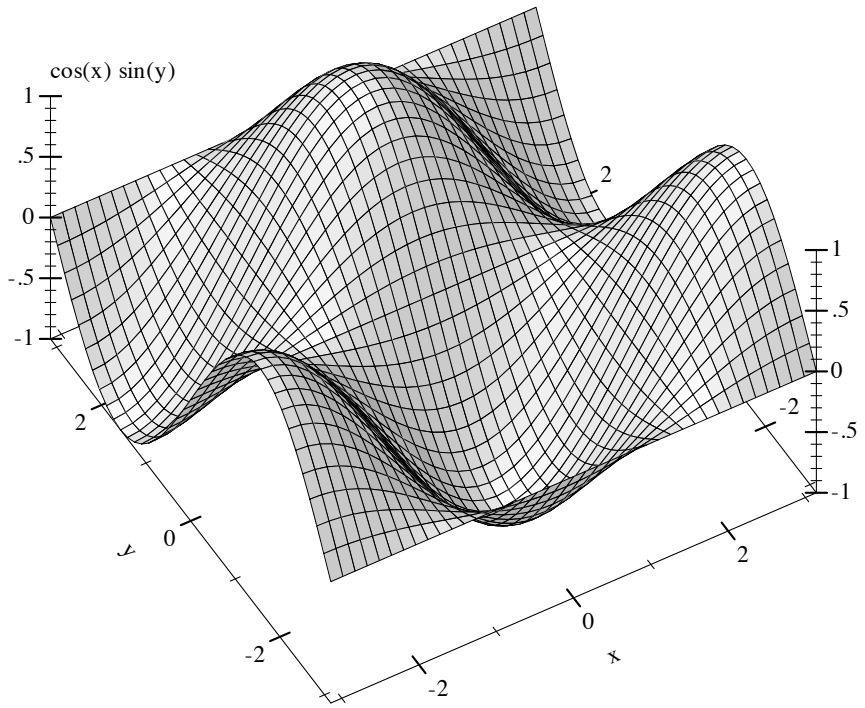
The `plot` function collects legend entries, bounds and ticks. It then sets up a *plot area* with large enough bounds to contain the renderers, draws the axes and ticks, invokes the renderers' drawing procedures, and then draws the legend.

## 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!

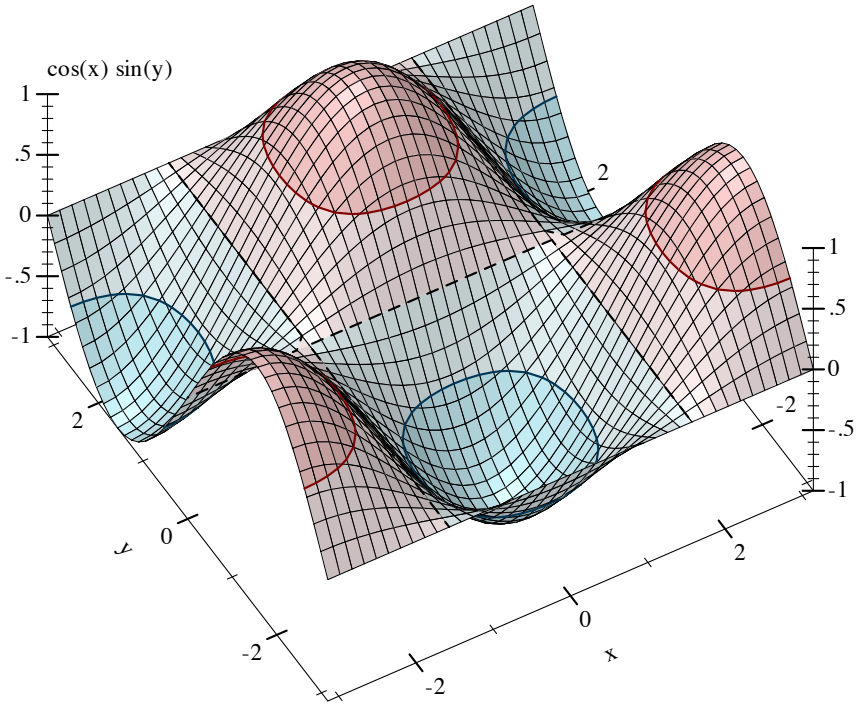



This example also demonstrates using keyword arguments that change the plot, such as `#:title`. In Plot, every keyword argument is optional and almost all have parameterized default values. In the case of `plot3d`'s `#:title`, the corresponding parameter is `plot-title`. That is, keyword arguments are 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)))
                                 (- pi) pi (- pi) pi)))
```

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 §8 "Plot and Renderer Parameters" for the full listing.

An R × R → R function

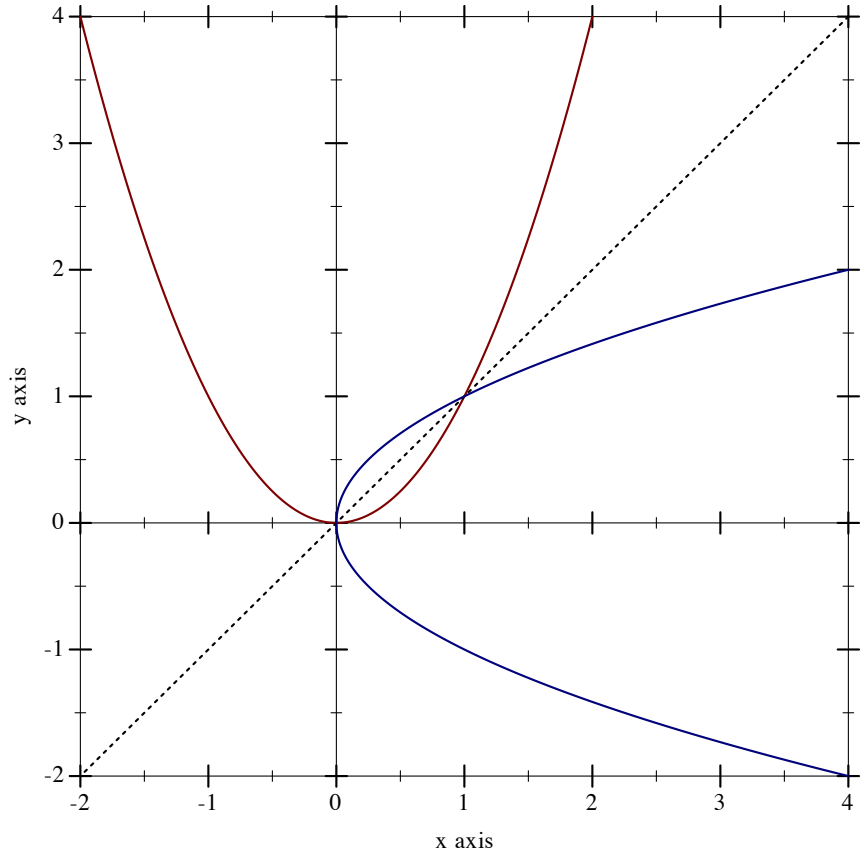



This example also demonstrates `contour-intervals3d`, which colors the surface between contour lines, or lines of constant height. By default, `contour-intervals3d` places the contour lines at the same heights as the ticks on the $z$ axis.

## 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.)

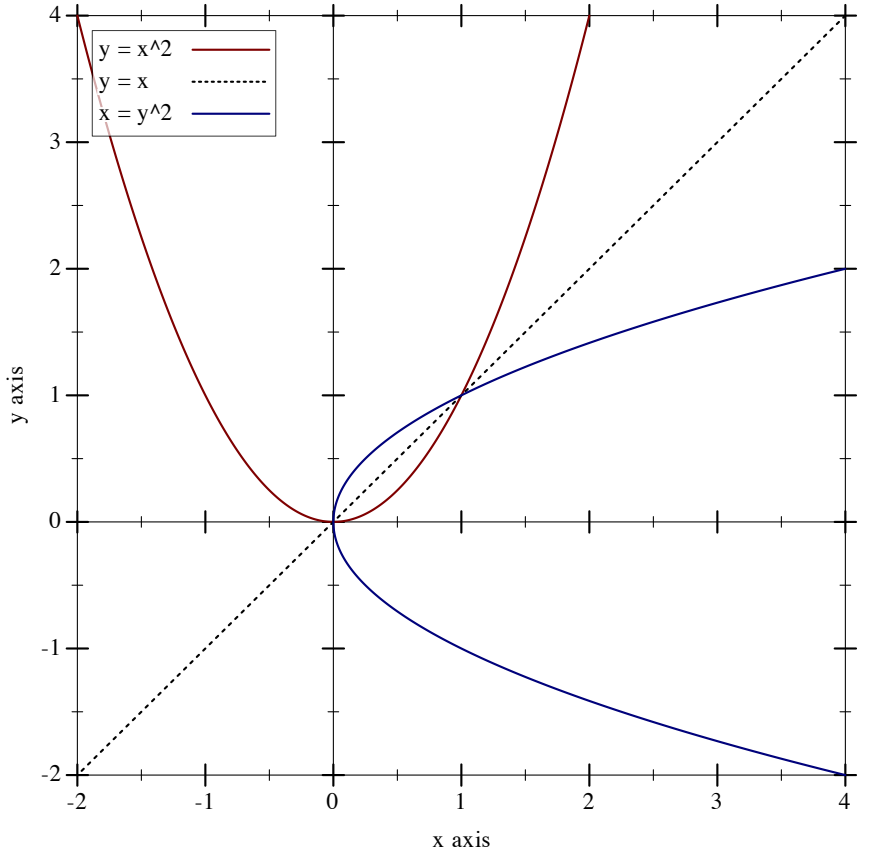
Notice the numbered colors. Plot additionally recognizes, as colors, lists of RGB values such as `'(128 128 0)`, `color%` instances, and strings like `"red"` and `"navajowhite"`. (The last are turned into RGB triples using a `color-database<%>`.) Use numbered colors when you just need different colors 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,

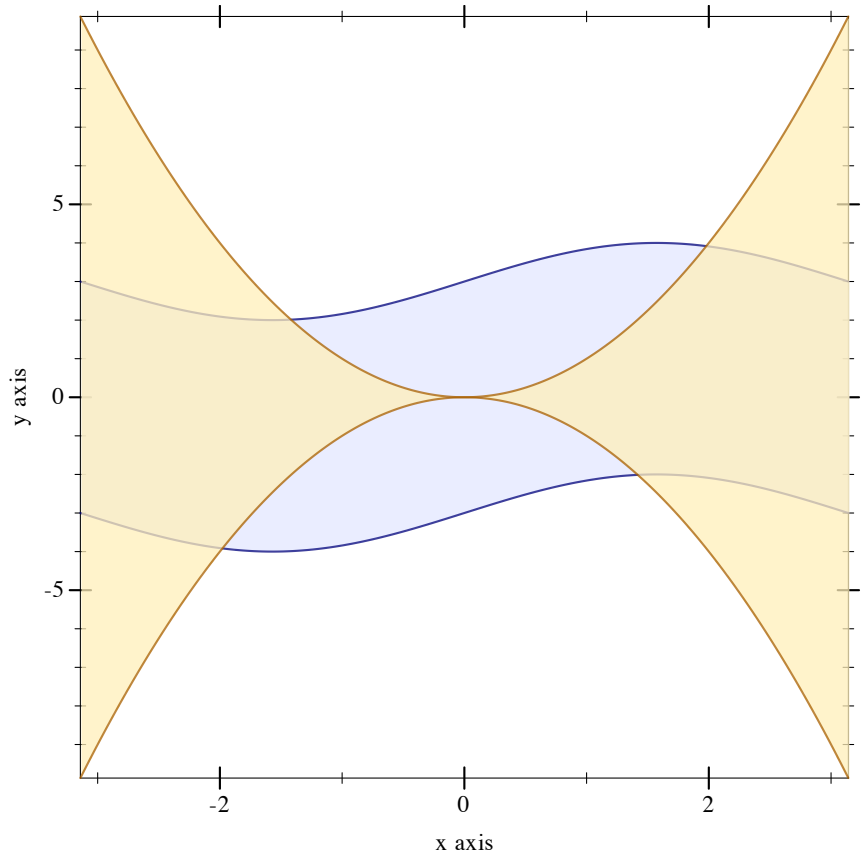
```
> (plot (list (axes)
              (function sqr -2 2 #:label "y = x^2")
              (function (λ (x) x) #:label "y =
```

```
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:

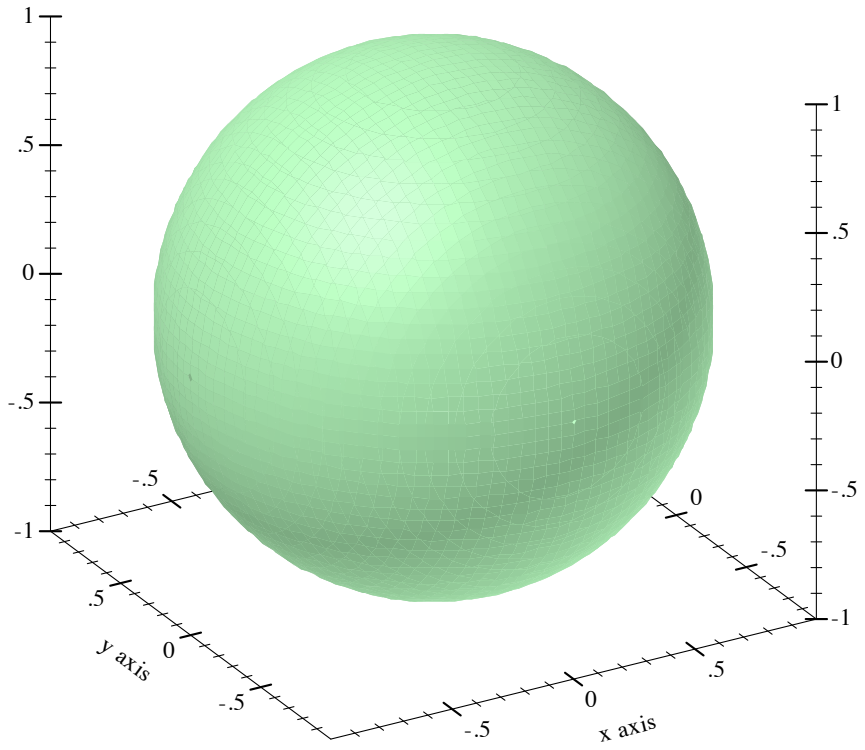
```
> (plot (function sqr))
plot: could not determine sensible plot bounds; got x ∈ [#f,#f], y ∈ [#f,#f]
> (plot (function sqr #f #f))
plot: could not determine sensible plot bounds; got x ∈ [#f,#f], y ∈ [#f,#f]
> (plot (function sqr -2))
```

*plot: could not determine sensible plot bounds; got x ∈ [-2,#f], y ∈ [#f,#f]*

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`:

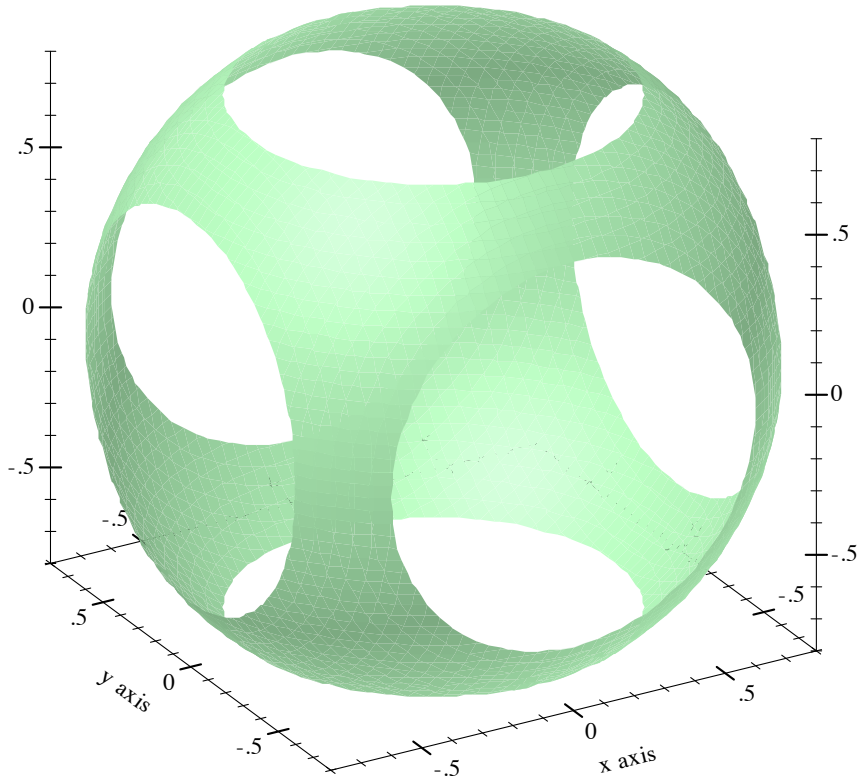
```
> (plot3d (polar3d (λ (θ ρ) 1) #:color 2 #:line-
style 'transparent)
          #:altitude 25)
```




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)
```

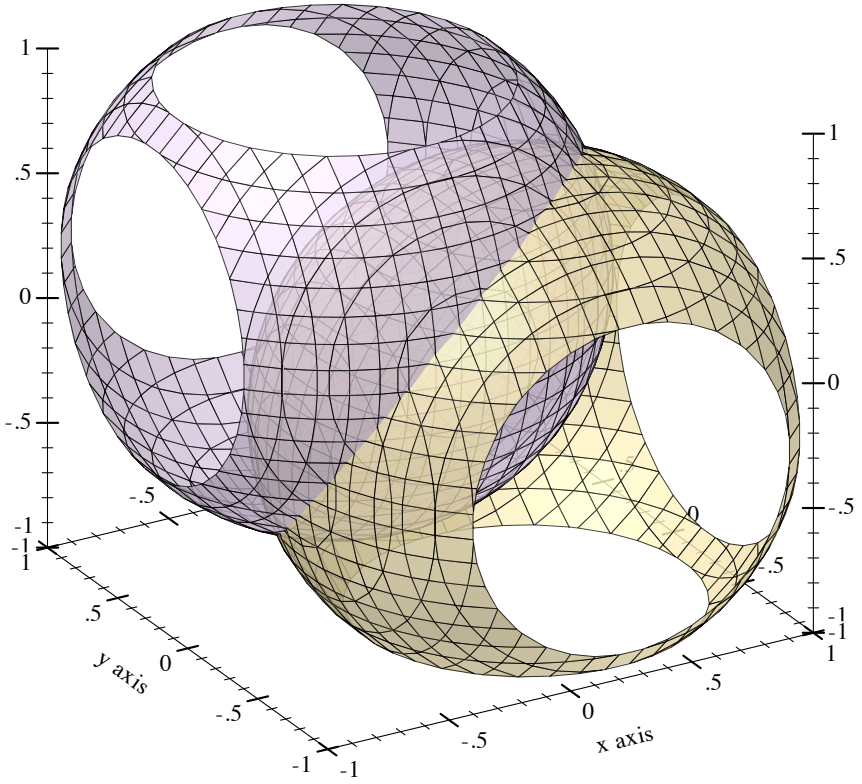



## 1.6 Plotting Multiple 3D Renderers

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

```
> (define ((dist cx cy cz) x y z)
    (sqrt (+ (sqr (- x cx)) (sqr (- y cy)) (sqr (- z cz)))))
> (plot3d (list (isosurface3d (dist  1/4 -1/4 -1/4) 0.995
                              #:color 4 #:alpha 0.8 #:samples 21)
                (isosurface3d (dist -1/4  1/4  1/4) 0.995
                              #:color 6 #:alpha 0.8 #:samples 21))
          #:x-min -1 #:x-max 1
          #:y-min -1 #:y-max 1
```

```
#:z-min -1 #:z-max 1
#:altitude 25)
```




## 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.

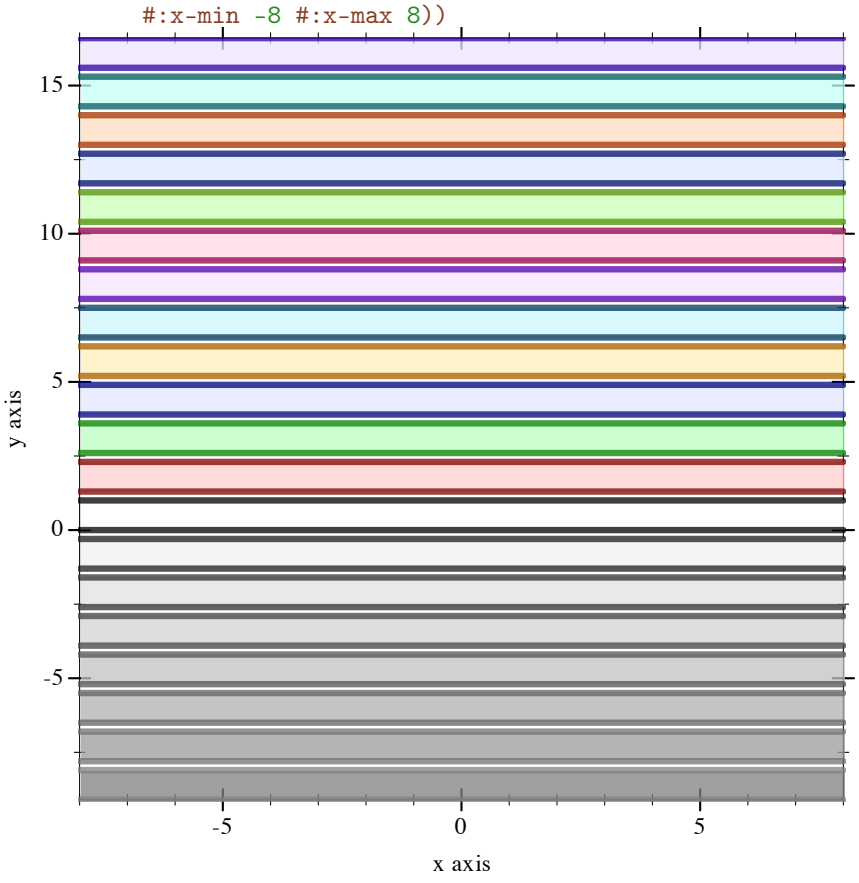
## 1.8 Colors and Styles

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))
```

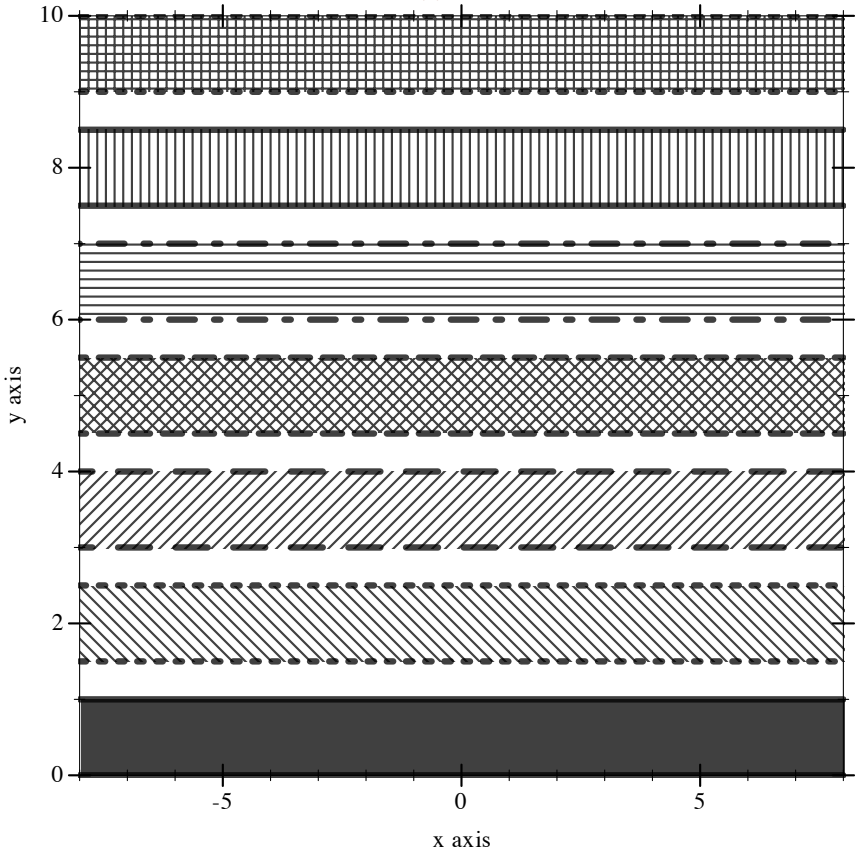



Color `0` is black for lines and white for areas. Colors `1`..`120` are generated by rotating hues and adjusting to make neighbors more visually dissimilar. Colors `121`..`127` are grayscale.

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 and 3D Plotting Procedures

The plotting procedures exported by `plot/no-gui` produce `bitmap%` and `pict` instances, and write to files. They do not require `racket/gui`, so they work in headless environments; for example, a Linux terminal with `DISPLAY` unset.

The `plot` module re-exports everything exported by `plot/no-gui`, as well as `plot`, `plot3d`, and other procedures that create interactive plots and plot frames. Interactive plotting procedures can always be imported, but fail when called if there is no working display or `racket/gui` is not present.

Each 3D plotting procedure behaves the same way as its corresponding 2D procedure, but takes the additional keyword arguments `#:z-min`, `#:z-max`, `#:angle`, `#:altitude` and `#:z-label`.

## 2.1 GUI Plotting Procedures

`(require plot)` package: `plot-gui-lib`

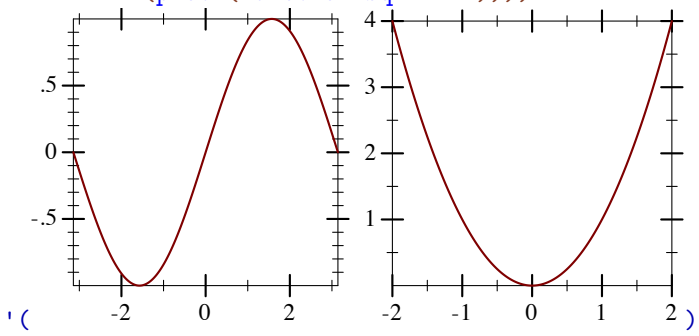
```
(plot renderer-tree
      [#:x-min x-min
       #:x-max x-max
       #:y-min y-min
       #:y-max y-max
       #:width width
       #:height height
       #:title title
       #:x-label x-label
       #:y-label y-label
       #:aspect-ratio aspect-ratio
       #:legend-anchor legend-anchor
       #:out-file out-file
       #:out-kind out-kind])
 → (or/c (and/c (is-a?/c snip%) (is-a?/c plot-metrics<%>)) 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? pict? #f) = (plot-title)
  x-label : (or/c string? pict? #f) = (plot-x-label)
  y-label : (or/c string? pict? #f) = (plot-y-label)
```

```
aspect-ratio : (or/c (and/c rational? positive?) #f)
             = (plot-aspect-ratio)
legend-anchor : legend-anchor/c = (plot-legend-anchor)
out-file : (or/c path-string? output-port? #f) = #f
out-kind : plot-file-format/c = '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))))
```




```
'(  )
```

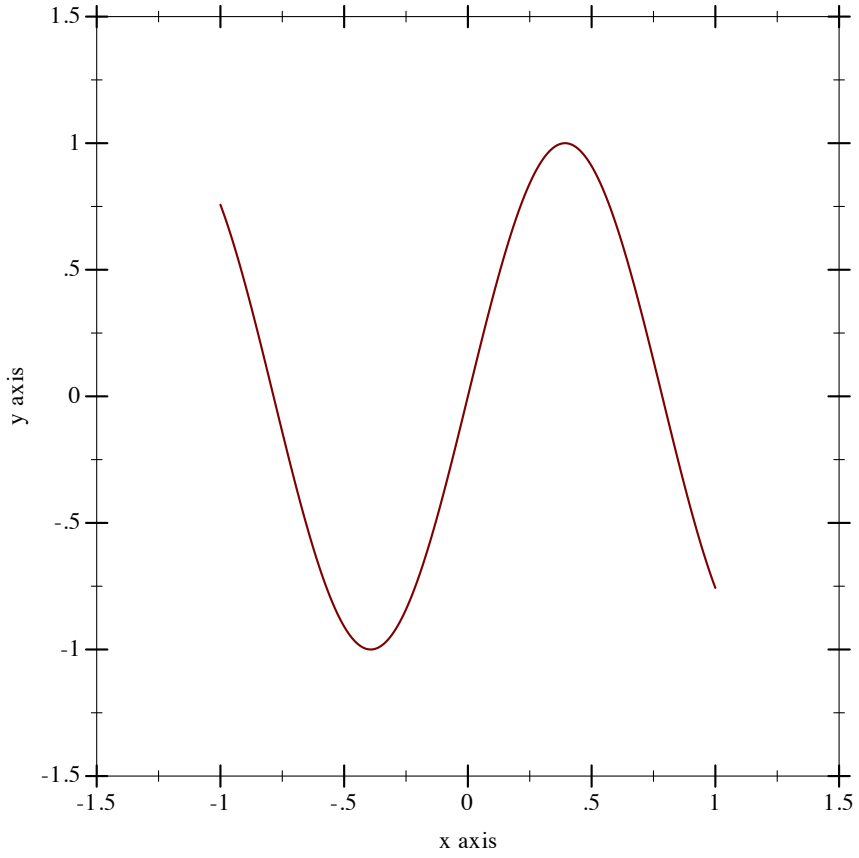
When the parameter `plot-new-window?` is `#t`, `plot` opens a new window to display the plot and returns `(void)`.

When `#:out-file` is given, `plot` writes the plot to a file using `plot-file` as well as returning a `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,

When given, the *aspect-ratio* argument defines the aspect ratio of the plot area, see `plot-aspect-ratio` for more details.

```
> (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.

Changed in version 7.9 of package `plot-gui-lib`: Added support for pictures for #:title, #:x-label and #:y-label. And to plot the legend outside the plot-area with #:legend-anchor

Changed in version 8.1 of package `plot-gui-lib`: Added #:aspect-ratio

```
(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
         #:aspect-ratio aspect-ratio
         #:legend-anchor legend-anchor
         #:out-file out-file
         #:out-kind out-kind])
 → (or/c (and/c (is-a?/c snip%) (is-a/c plot-metrics<%>)) 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? pict? #f) = (plot-title)
  x-label : (or/c string? pict? #f) = (plot-x-label)
  y-label : (or/c string? pict? #f) = (plot-y-label)
  z-label : (or/c string? pict? #f) = (plot-z-label)
  aspect-ratio : (or/c (and/c rational? positive?) #f)
               = (plot-aspect-ratio)
  legend-anchor : legend-anchor/c = (plot-legend-anchor)
  out-file : (or/c path-string? output-port? #f) = #f
  out-kind : plot-file-format/c = 'auto
```

Plots a 3D renderer or list of renderers (or more generally, a tree of renderers), as returned by `points3d`, `parametric3d`, `surface3d`, `isosurface3d`, 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 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.

When given, the *aspect-ratio* argument defines the aspect ratio of the plot area, see plot-aspect-ratio for more details.

**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.

Changed in version 7.9 of package plot-gui-lib: Added support for pictures for #:title, #:x-label and #:y-label. And to plot the legend outside the plot-area with #:legend-anchor

Changed in version 8.1 of package plot-gui-lib: Added #:aspect-ratio

```
(plot-snip <plot-argument> ...)
 → (and/c (is-a?/c 2d-plot-snip%) (is-a?/c plot-metrics<%>))
  <plot-argument> : <plot-argument-contract>
```

```
(plot3d-snip <plot-argument> ...)
 → (and/c (is-a?/c snip%) (is-a?/c plot-metrics<%>))
  <plot-argument> : <plot-argument-contract>
```

```
(plot-frame <plot-argument> ...) → (is-a?/c frame%)
  <plot-argument> : <plot-argument-contract>
```

```
(plot3d-frame <plot-argument> ...) → (is-a?/c frame%)
  <plot-argument> : <plot-argument-contract>
```

Plot to different GUI backends. These procedures accept the same arguments as plot and plot3d, except deprecated keywords, and #:out-file and #:out-kind.

Use plot-frame and plot3d-frame to create a frame% regardless of the value of plot-new-window?. The frame is initially hidden.

Use plot-snip and plot3d-snip to create an interactive snip% regardless of the value of plot-new-window?.

The snip% objects returned by plot-snip can be used to construct interactive plots. See §3.9 “Interactive Overlays for 2D plots” for more details.

## 2.2 Non-GUI Plotting Procedures

```
(require plot/no-gui)
```
package: plot-lib

```
(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 : plot-file-format/c = 'auto
  <plot-keyword> : <plot-keyword-contract>
```

```
(plot3d-file renderer-tree
             output
            [kind]
             #:<plot3d-keyword> <plot3d-keyword> ...) → void?
  renderer-tree : (treeof (or/c renderer3d? nonrenderer?))
  output : (or/c path-string? output-port?)
  kind : plot-file-format/c = 'auto
  <plot3d-keyword> : <plot3d-keyword-contract>
```

```
(plot-pict <plot-argument> ...) → plot-pict?
  <plot-argument> : <plot-argument-contract>
```

```
(plot3d-pict <plot3d-argument> ...) → plot-pict?
  <plot3d-argument> : <plot3d-argument-contract>
```

```
(plot-bitmap <plot-argument> ...)
 → (and/c (is-a?/c bitmap%) (is-a?/c plot-metrics<%>))
  <plot-argument> : <plot-argument-contract>
```

```
(plot3d-bitmap <plot3d-argument> ...)
 → (and/c (is-a?/c bitmap%) (is-a?/c plot-metrics<%>))
  <plot3d-argument> : <plot3d-argument-contract>
```

Plot to different non-GUI backends. These procedures accept the same arguments as `plot` and `plot3d`, except deprecated keywords, and `#:out-file` and `#:out-kind`.

Use `plot-file` or `plot3d-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 to set printing parameters. (See `post-script-dc%` and `pdf-dc%`.) When `kind` is `'auto`, `plot-file` and `plot3d-file` try to determine from the file name extension the kind of file to write.

Use `plot-pict` or `plot3d-pict` to create a `pict`. For example, this program creates a slide containing a 2D plot of a parabola:

```
#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")))
```

Use `plot-bitmap` or `plot3d-bitmap` to create a `bitmap%`.

```
(plot/dc renderer-tree
         dc
         x
         y
         width
         height
         #:<plot-keyword> <plot-keyword> ...)
 → (is-a?/c plot-metrics<%>)
  renderer-tree : (treeof (or/c renderer2d? nonrenderer?))
  dc : (is-a?/c dc<%>)
  x : real?
  y : real?
  width : (>=/c 0)
  height : (>=/c 0)
  <plot-keyword> : <plot-keyword-contract>
```

```
(plot3d/dc renderer-tree
           dc
           x
           y
           width
           height
           #:<plot3d-keyword> <plot3d-keyword> ...)
 → (is-a?/c plot-metrics<%>)
  renderer-tree : (treeof (or/c renderer3d? nonrenderer?))
  dc : (is-a?/c dc<%>)
  x : real?
  y : real?
  width : (>=/c 0)
  height : (>=/c 0)
  <plot3d-keyword> : <plot3d-keyword-contract>
```

Plot to an arbitrary device context, in the rectangle with width *width*, height *height*, and upper-left corner *x*,*y*. These procedures accept the same arguments as `plot` and `plot3d`, except deprecated keywords, and `#:out-file` and `#:out-kind`.

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

## 2.3 Pict-Plotting Work-a-Likes

```
(require plot/pict)
```
package: `plot-lib`

When setting up an evaluator for a Scribble manual, require `plot/pict` instead of `plot`. Evaluation will produce picts instead of snips, which scale nicely in PDF-rendered documentation.

For example, this is how the evaluator for the Plot documentation is defined:

```
(define plot-eval
  (let ([eval  (make-base-eval)])
    (eval '(begin
             (require racket/math
                      racket/match
                      racket/list
                      racket/draw
                      racket/class
                      plot/pict
                      plot/utils)))
    eval))
```

If you use (require (for-label plot)), links in example code should resolve to documentation for the functions exported by plot.

```
(plot <plot-argument> ...) → pict?
  <plot-argument> : <plot-argument-contract>
```

```
(plot3d <plot3d-argument> ...) → pict?
  <plot3d-argument> : <plot3d-argument-contract>
```

Like the functions of the same name exported from plot, but these produce pict instances instead of interactive snips.

## 2.4 Bitmap-Plotting Work-a-Likes

(require plot/bitmap) package: plot-lib

When plotting in an environment where bitmap% instances can be shown but snip% instances cannot (for example, on a web page that evaluates Racket code), require plot/bitmap instead of plot.

```
(plot <plot-argument> ...) → (is-a?/c bitmap%)
  <plot-argument> : <plot-argument-contract>
```

```
(plot3d <plot3d-argument> ...) → (is-a?/c bitmap%)
  <plot3d-argument> : <plot3d-argument-contract>
```

Like the functions of the same name exported from plot, but these produce bitmap% instances instead of interactive snips.

# 3 2D Renderers

`(require plot)` package: `plot-gui-lib`

## 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. The label argument can be a string or a `pict`. For most use cases, the string will be sufficient, especially since it allows using Unicode characters, and thus some mathematical notation. For more complex cases, a `pict` can be used, whic allows arbitrary text and graphics to be used as label entries.

Changed in version 7.9 of package `plot-gui-lib`: Added support for pictures for #:label

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

## 3.2 2D Point Renderers

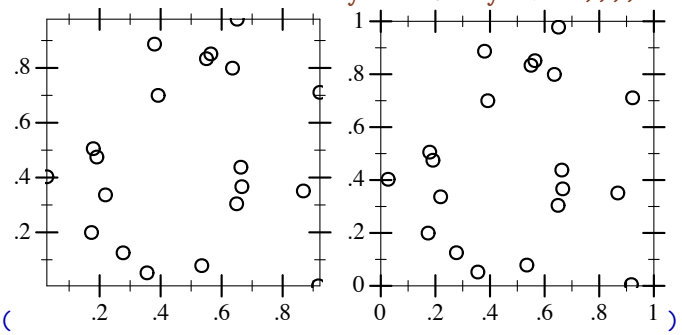
```
(points  vs
         [#:x-min x-min
          #:x-max x-max
          #:y-min y-min
          #:y-max y-max
          #:sym sym
          #:color color
          #:fill-color fill-color
          #:x-jitter x-jitter
          #:y-jitter y-jitter
          #:size size
          #:line-width line-width
          #:alpha alpha
          #:label label])          → renderer2d?
  vs : (sequence/c (sequence/c #:min-count 2 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
  x-jitter : (>=/c 0) = (point-x-jitter)
  y-jitter : (>=/c 0) = (point-y-jitter)
  size : (>=/c 0) = (point-size)
  line-width : (>=/c 0) = (point-line-width)
```

```
alpha : (real-in 0 1) = (point-alpha)
label : (or/c string? pict? #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))))
```




'(  )

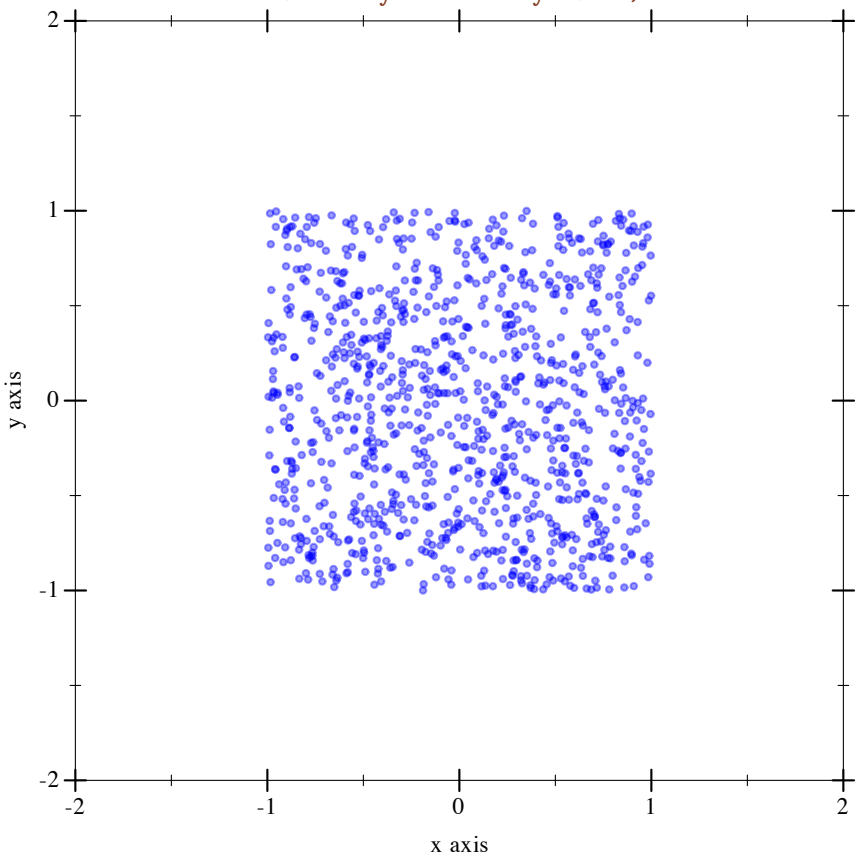
Readers of the first plot could only guess that the random points were generated in [0,1] × [0,1].

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.

When *x-jitter* is non-zero, all points are translated by a random amount at most *x-jitter* from their original position along the x-axis. A non-zero *y-jitter* similarly translates points along the y-axis. Jitter is added in both directions so total spread is twice the value given. To be precise, each point `p` is moved to a random location inside a rectangle centered at `p` with width at most twice *x-jitter* and height at most twice *y-jitter* subject to the constraint that new points lie within [*x-min*, *x-max*] and [*y-min*, *y-max*] if these bounds are non-`#f`.

```
> (plot
```

```
(points (for/list ([i (in-range 1000)])
          (list 0 0))
        #:alpha 0.4
        #:x-jitter 1
        #:y-jitter 1
        #:sym 'fullcircle1
        #:color "blue")
#:x-min -2 #:x-max 2 #:y-min -2 #:y-max 2)
```




Randomly moving data points is almost always a bad idea, but jittering in a controlled manner can sometimes be useful. For example:

More examples of jittering: Another Look at the California Vaccination Data and Typing with Pleasure

- To highlight the size of a dense (or overplotted) sample.
- To see the distribution of 1-dimensional data; as a substitute for box or violin plots.
- To anonymize spatial data, showing i.e. an office's neighborhood but hiding its address.

Changed in version 7.9 of package `plot-gui-lib`: Added support for pictures for #:label

```
(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? (sequence/c real?))
            (-> (vector/c real? real?) (sequence/c 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? pict? #f) = #f
```

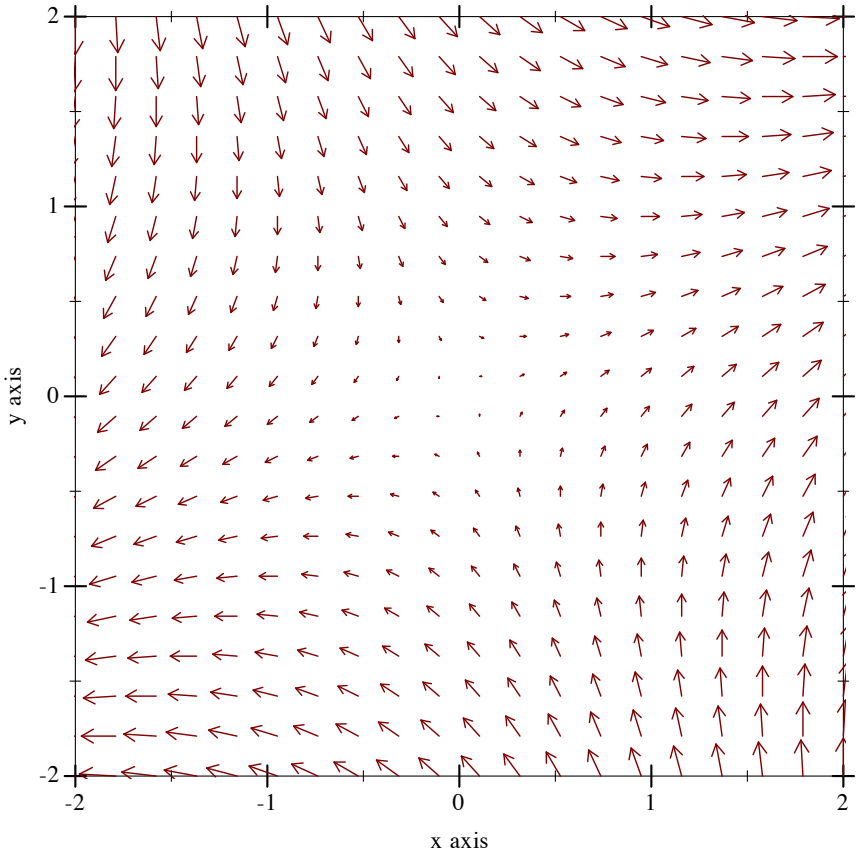
Returns a renderer that draws a vector field.

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`.

The shape of the arrow-head can be controlled with `arrow-head-size-or-scale` and `arrow-head-angle`.

An example of automatic scaling:

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



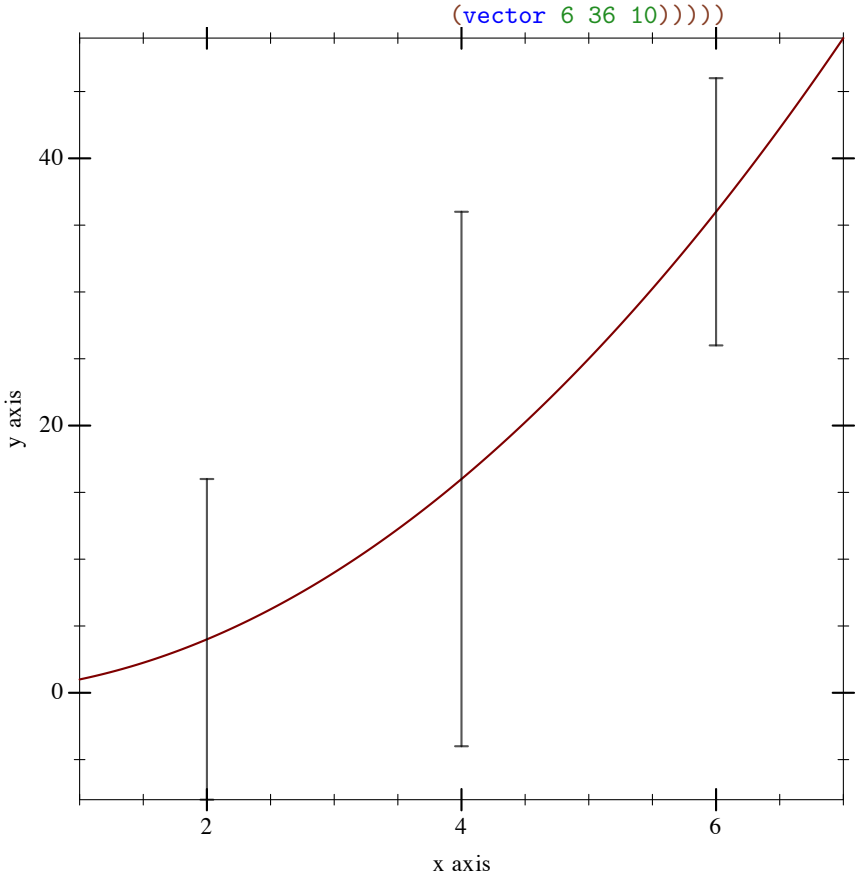

Changed in version 7.9 of package `plot-gui-lib`: Added support for pictures for #:label and controlling the arrowhead

```
(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
              #:invert? invert?])      → renderer2d?
  bars : (sequence/c (sequence/c #:min-count 3 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)
invert? : boolean? = #f
```

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)))))
```



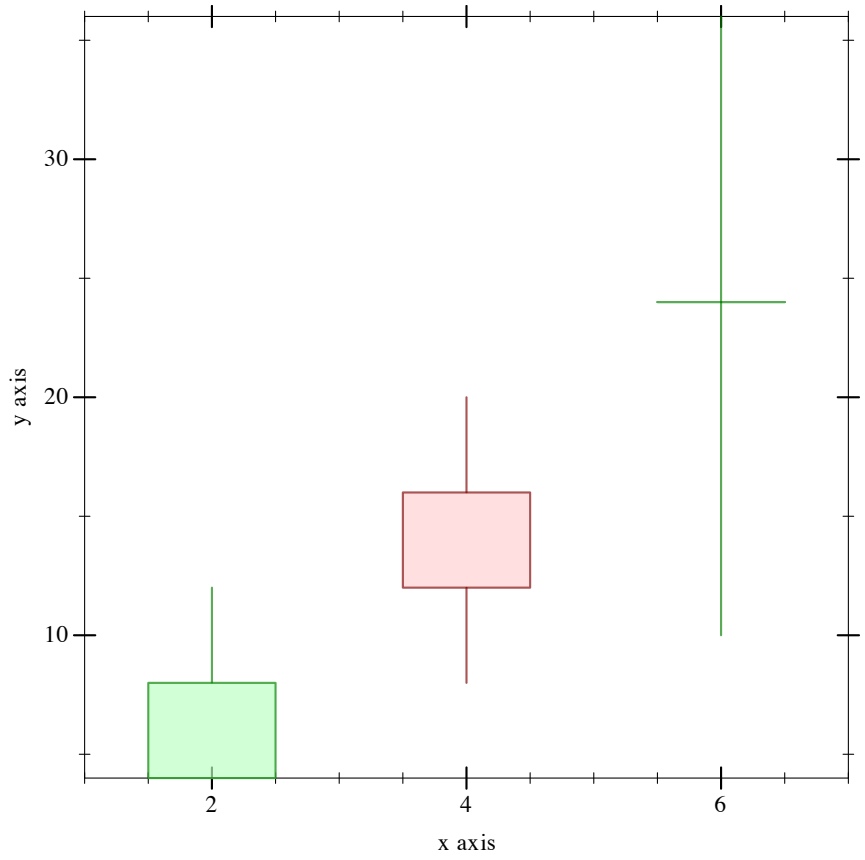

If *invert?* is #t, the x and y coordinates are inverted, and the bars are drawn horizontally rather than vertically. This is intended for use with the corresponding option of discrete-histogram and stacked-histogram.

Changed in version 1.1 of package `plot-gui-lib`: Added the `#:invert?` option.

```
(candlesticks  candles
               [#:x-min x-min
                #:x-max x-max
                #:y-min y-min
                #:y-max y-max
                #:up-color up-color
                #:down-color down-color
                #:line-width line-width
                #:line-style line-style
                #:width width
                #:alpha alpha])          → renderer2d?
  candles : (sequence/c (sequence/c #:min-count 5 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
  up-color : plot-color/c = (candlestick-up-color)
  down-color : plot-color/c = (candlestick-down-color)
  line-width : (>=/c 0) = (candlestick-line-width)
  line-style : plot-pen-style/c = (candlestick-line-style)
  width : (>=/c 0) = (candlestick-width)
  alpha : (real-in 0 1) = (candlestick-alpha)
```

Returns a renderer that draws candlesticks. This is most common when plotting historical prices for financial instruments. The first element in each vector of *candles* comprises the x-axis coordinate; the second, third, fourth, and fifth elements in each vector comprise the open, high, low, and close, respectively, of the y-axis coordinates.

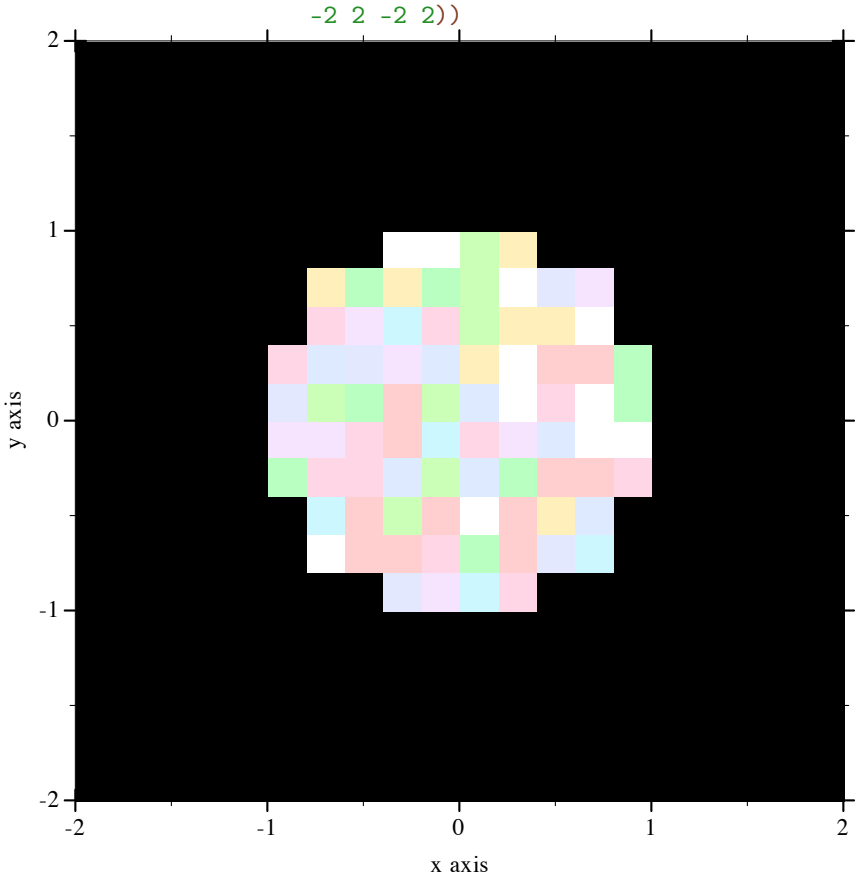
```
> (plot (list (candlesticks (list (vector 2 4 12 4 8)
                                  (vector 4 16 20 8 12)
                                  (vector 6 24 36 10 24)))))
```




```
(color-field  f
              [x-min
               x-max
               y-min
               y-max
               #:samples samples
               #:alpha alpha])    → renderer2d?
  f : (or/c (-> real? real? plot-color/c)
            (-> (vector/c real? real?) plot-color/c))
  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? = (color-field-samples)
  alpha : (real-in 0 1) = (color-field-alpha)
```

Returns a renderer that draws rectangles filled with a color based on the center point.

```
> (plot (color-field (λ (x y) (if (< (+ (sqr x) (sqr y)) 1) (random 10) 'black))
                     -2 2 -2 2))
```



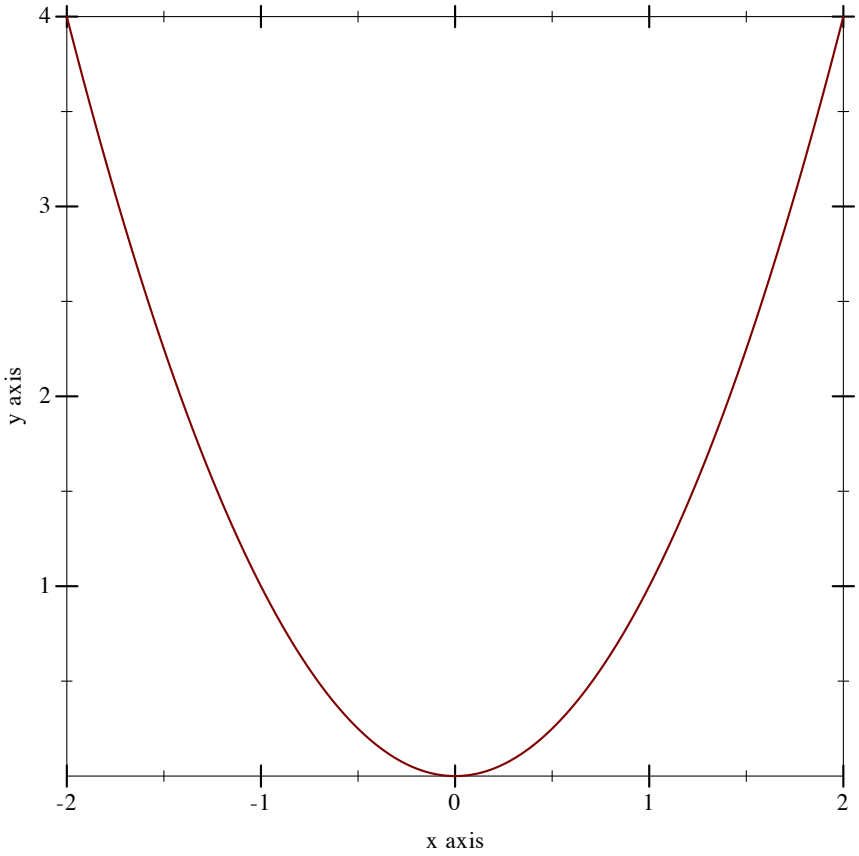

Added in version 7.9 of package `plot-gui-lib`.

## 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
            #:marker marker
            #:marker-color marker-color
            #:marker-fill-color marker-fill-color
            #:marker-size marker-size
            #:marker-line-width marker-line-width
            #:marker-alpha marker-alpha
            #:marker-count marker-count
            #: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)
  marker : point-sym/c = 'none
  marker-color : (or/c 'auto plot-color/c) = 'auto
  marker-fill-color : (or/c 'auto plot-color/c) = 'auto
  marker-size : (>=/c 0) = (point-size)
  marker-line-width : (>=/c 0) = (point-line-width)
  marker-alpha : (real-in 0 1) = (point-alpha)
  marker-count : positive-integer? = 20
  label : (or/c string? pict? #f) = #f
```

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

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




When *marker* is not 'none, markers will be placed on the on the line drawn for the function at equal intervals. The marker and related arguments are the same as for the points renderer. The number of markers shown on the plot is specified by *marker-count* parameter.

Using markers has the same effect as using both a function and a points renderer in a single plot, exept that using *marker*s will show the marker super-imposed over the line style in the plot legend.

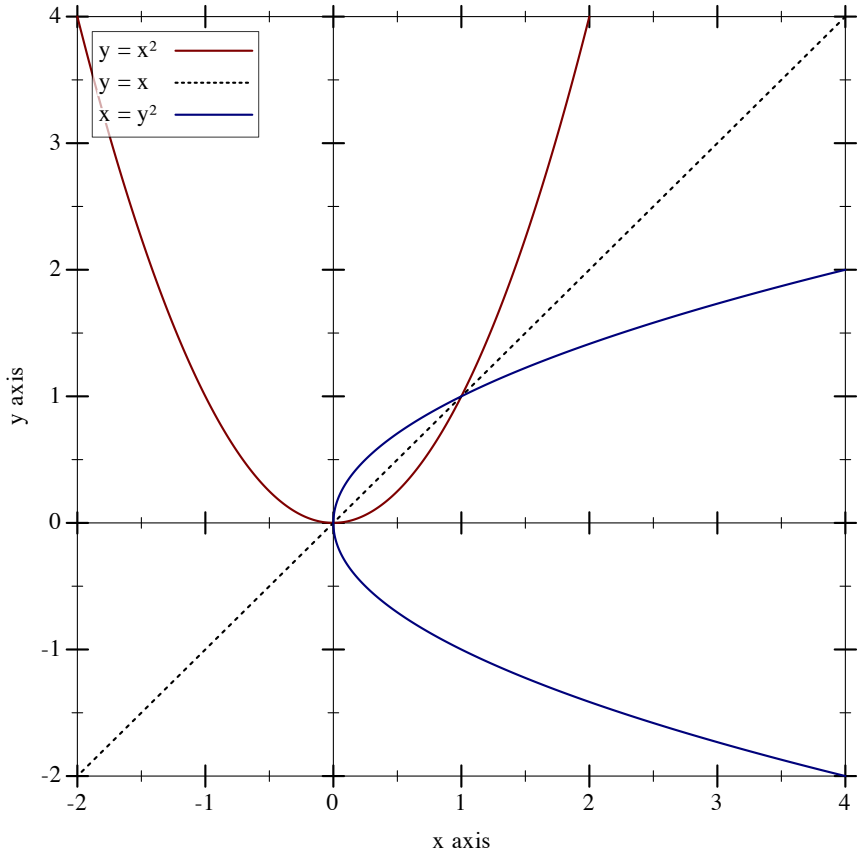
Changed in version 7.9 of package plot-gui-lib: #:label argument supports pictures

Changed in version 8.10 of package plot-gui-lib: #:marker and related arguments added

```
(inverse  f
          [y-min
           y-max
           #:x-min x-min
           #:x-max x-max
           #:samples samples
           #:color color
           #:width width
           #:style style
           #:alpha alpha
           #:marker marker
           #:marker-color marker-color
           #:marker-fill-color marker-fill-color
           #:marker-size marker-size
           #:marker-line-width marker-line-width
           #:marker-alpha marker-alpha
           #:marker-count marker-count
           #: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)
  marker : point-sym/c = 'none
  marker-color : (or/c 'auto plot-color/c) = 'auto
  marker-fill-color : (or/c 'auto plot-color/c) = 'auto
  marker-size : (>=/c 0) = (point-size)
  marker-line-width : (>=/c 0) = (point-line-width)
  marker-alpha : (real-in 0 1) = (point-alpha)
  marker-count : positive-integer? = 20
  label : (or/c string? pict? #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")))
```




The *marker* and related arguments have the same meaning as for the function renderer.

Changed in version 7.9 of package plot-gui-lib: #:label argument supports pictures

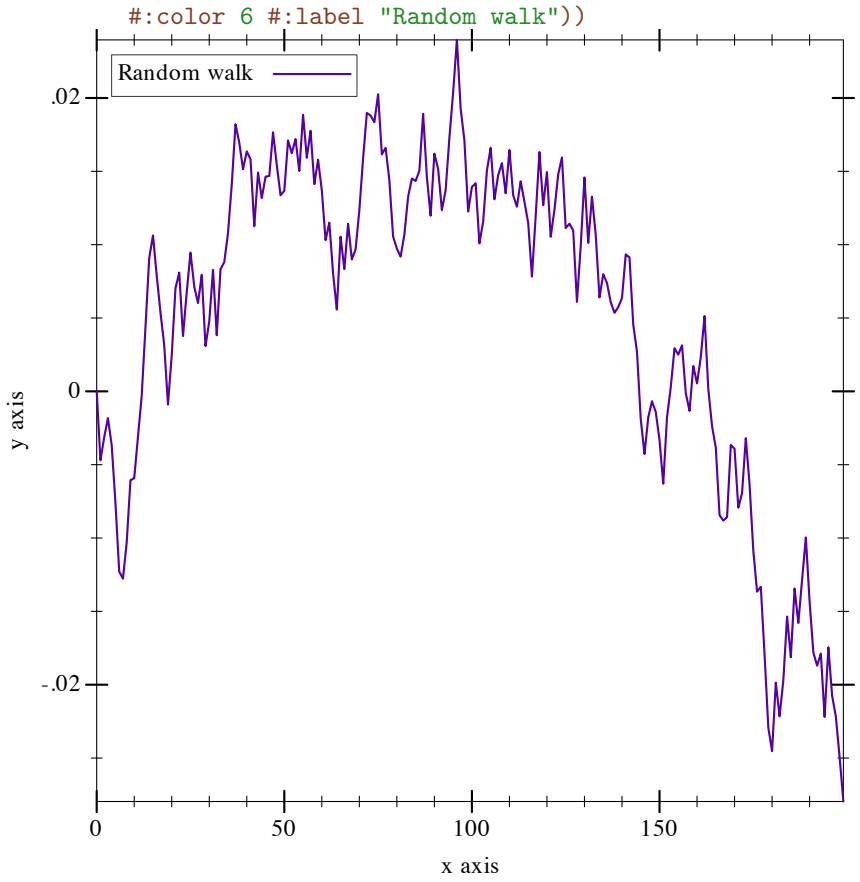
Changed in version 8.10 of package plot-gui-lib: #:marker and related arguments added

```
(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
        #:marker marker
        #:marker-color marker-color
        #:marker-fill-color marker-fill-color
        #:marker-size marker-size
        #:marker-line-width marker-line-width
        #:marker-alpha marker-alpha
        #:label label
        #:ignore-axis-transforms? ignore-axis-transforms?])
 → renderer2d?
  vs : (sequence/c (sequence/c #:min-count 2 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)
  marker : point-sym/c = 'none
  marker-color : (or/c 'auto plot-color/c) = 'auto
  marker-fill-color : (or/c 'auto plot-color/c) = 'auto
  marker-size : (>=/c 0) = (point-size)
  marker-line-width : (>=/c 0) = (point-line-width)
  marker-alpha : (real-in 0 1) = (point-alpha)
  label : (or/c string? pict? #f) = #f
  ignore-axis-transforms? : boolean? = #f
```

Returns a renderer that draws lines connecting the points in the input sequence *vs*.

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)))
```




If any of the points in *vs* is `+nan.0`, no line segment will be drawn at that position. This can be used to draw several independent data sets with one `lines` renderer, improving rendering performence for large datasets.

When *marker* is not `'none`, markers will be placed on the on the line at each point in *vs*. The marker and related arguments are the same as for the `points` renderer.

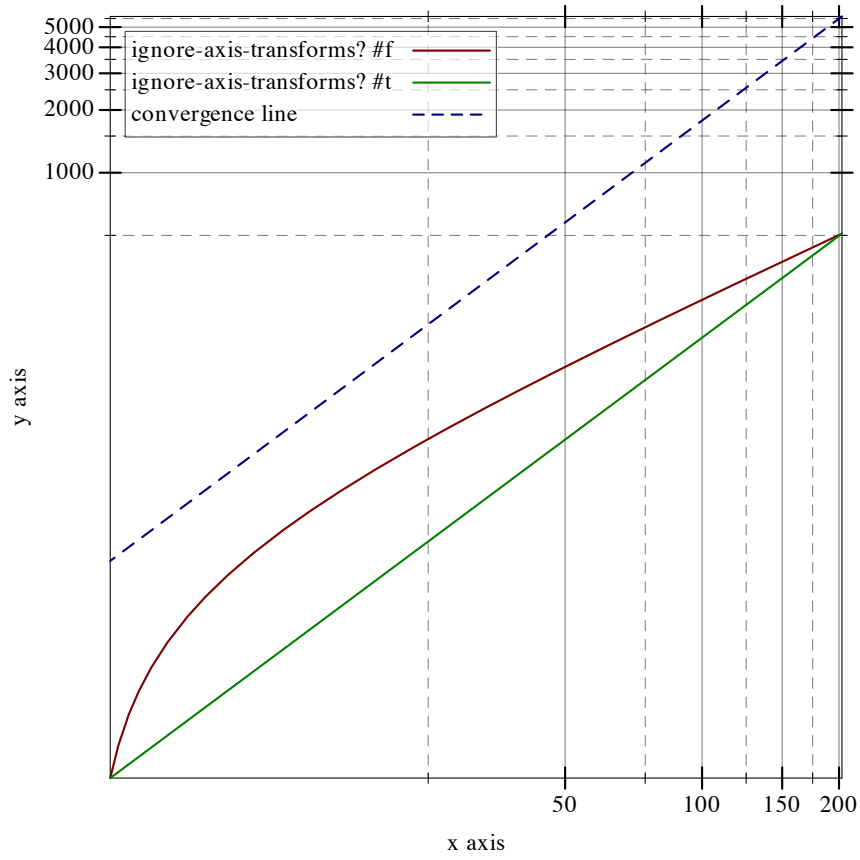
Using markers has the same effect as using both a `lines` and a `points` renderer in a single plot, exept that using *marker*s will show the marker super-imposed over the line style in the plot legend.

When *ignore-axis-transforms?* is `#t`, only the individual points in *vs* are affected by axis transforms, not the lines that connect these points. This feature can be used, for example, to plot a convergence line on a log-log plot.

**NOTE:** It is undesirable to ignore axis transforms in plots, but this feature can be used to replicate functionality of other plotting libraries and it is meant for users familiar with those

libraries. In Racket, it is preferable show the convergence line on log-log plots using the `function` renderer with the power function based on slope and intercept.

```
> (let ([data '((5 1.24) (203 510))]
        [slope 1.6252]
        [intercept -2.4005])
    (parameterize ([plot-x-transform log-transform]
                   [plot-y-transform log-transform])
      (plot
        (list
          (tick-grid)
          (lines data
                 #:ignore-axis-transforms? #f
                 #:label "ignore-axis-transforms? #f"
                 #:color 1)
          (lines data
                 #:ignore-axis-transforms? #t
                 #:label "ignore-axis-transforms? #t"
                 #:color 2)
          (function (lambda (x) (* (exp intercept)) (expt x slope))
                    #:style 'long-dash
                    #:label "convergence line"
                    #:color 3)))))
```




Changed in version 7.9 of package `plot-gui-lib`: #:label argument supports pictures

Changed in version 8.9 of package `plot-gui-lib`: #:ignore-axis-transforms? argument added

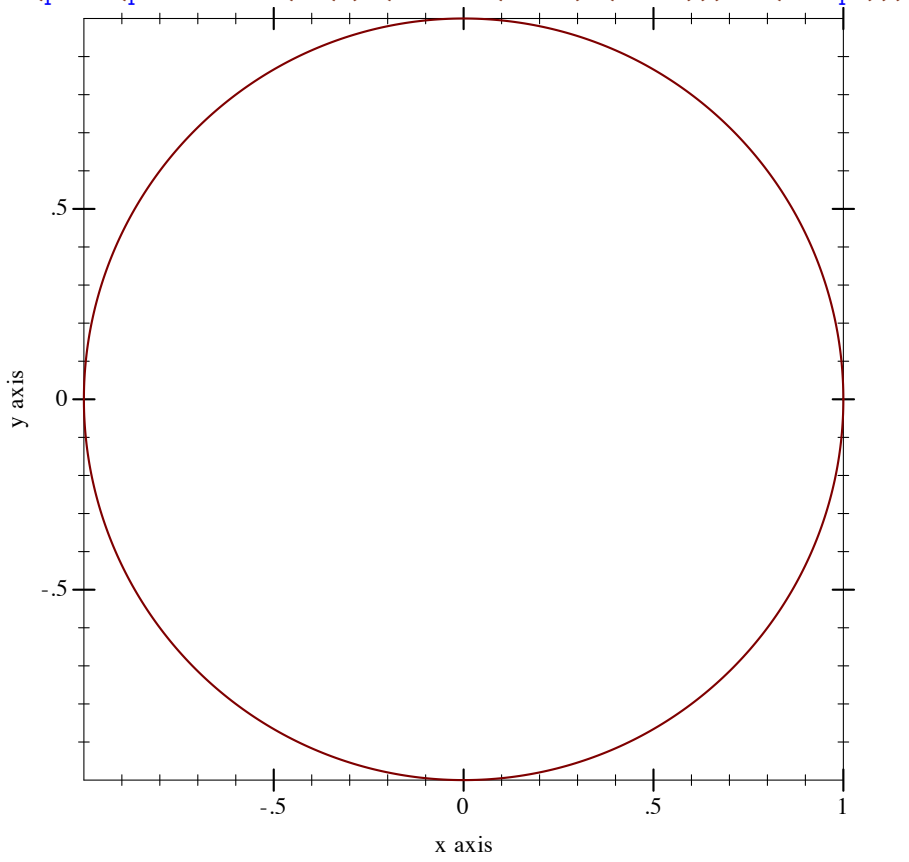
Changed in version 8.10 of package `plot-gui-lib`: #:marker and related arguments added

```
(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? . -> . (sequence/c 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? pict? #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)))
```



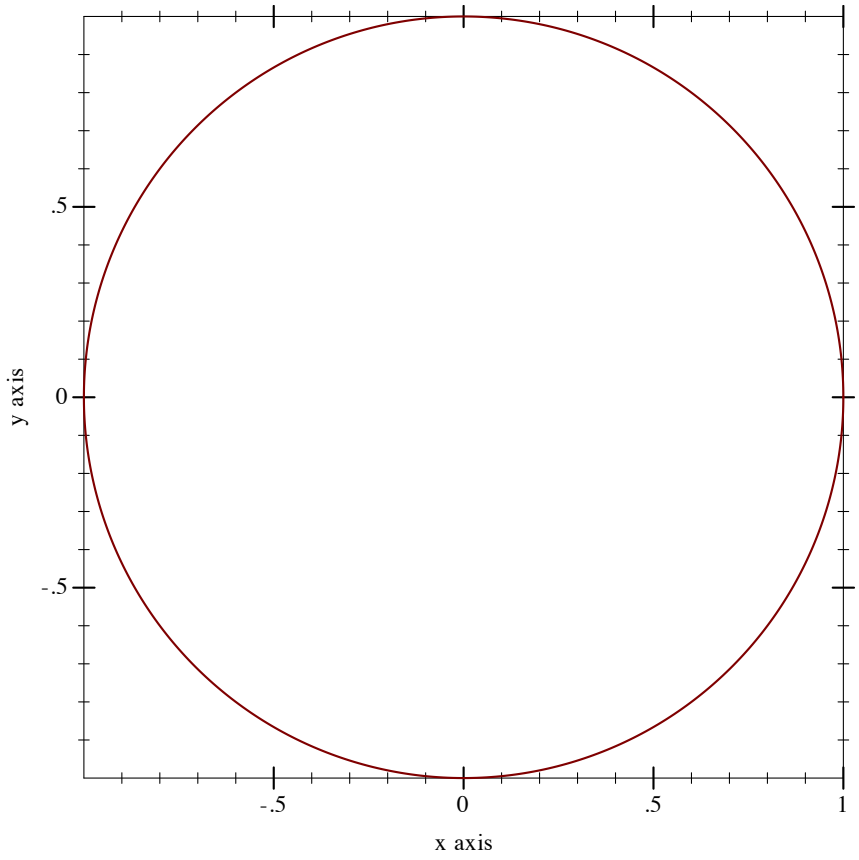
Changed in version 7.9 of package `plot-gui-lib`: Added support for pictures for #:label

```
(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? pict? #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)))
```




Changed in version 7.9 of package `plot-gui-lib`: Added support for pictures for #:label

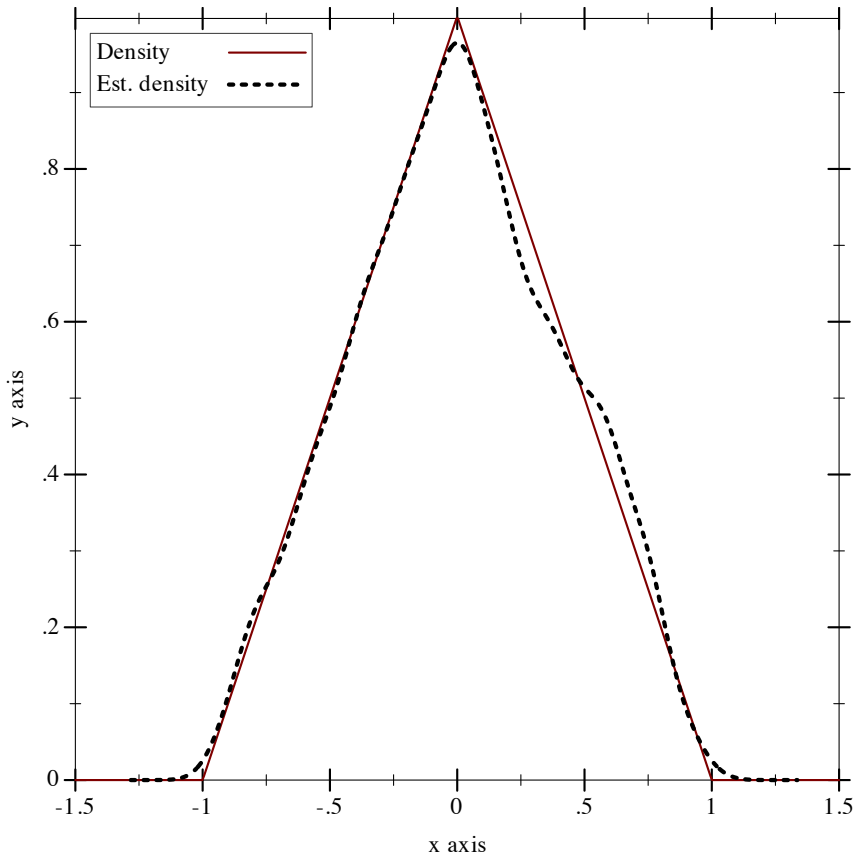
```
(density  xs
          [bw-adjust
           ws
           #:x-min x-min
           #:x-max x-max
           #:y-min y-min
           #:y-max y-max
           #:samples samples
           #:color color
           #:width width
           #:style style
           #:alpha alpha
           #:label label])    → renderer2d?
  xs : (sequence/c real?)
  bw-adjust : (>/c 0) = 1
  ws : (or/c (sequence/c (>=/c 0)) #f) = #f
```

```
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? pict? #f) = #f
```

Returns a renderer that plots an estimated density for the given points, which are optionally weighted by *ws*. 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")))
```




Changed in version 7.9 of package `plot-gui-lib`: Added support for pictures for #:label

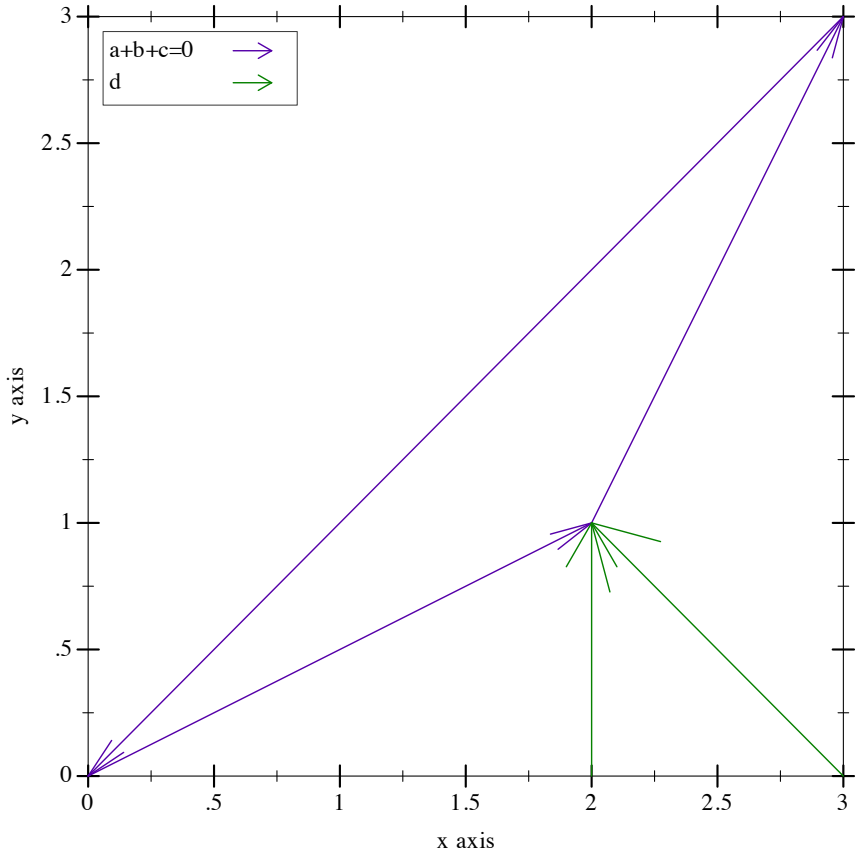
```
(arrows  vs
         [#:x-min x-min
          #:x-max x-max
          #:y-min y-min
          #:y-max y-max
          #:color color
          #:width width
          #:style style
          #:alpha alpha
          #:arrow-head-size-or-scale size
          #:arrow-head-angle angle
          #:label label])                  → renderer2d?
  vs : (or/c (listof (sequence/c #:min-count 2 real?))
             (vectorof (vector/c (sequence/c #:min-count 2 real?)
                                 (sequence/c #:min-count 2 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 = (arrows-color)
  width : (>=/c 0) = (arrows-line-width)
  style : plot-pen-style/c = (arrows-line-style)
  alpha : (real-in 0 1) = (arrows-alpha)
  size : (or/c (list/c '= (>=/c 0)) (>=/c 0))
       = (arrow-head-size-or-scale)
  angle : (>=/c 0) = (arrow-head-angle)
  label : (or/c string? pict? #f) = #f
```

Returns a renderer which draws arrows. Arrows can be specified either as sequences of 2D points, in this case they will be drawn as connected arrows between each two adjacent points, or they can be specified as an origin point and a rectangular magnitude vector, in which case each arrow is drawn individually. See example below.

In *vs* list and vector are interchangeable. Arrow-heads are only drawn when the endpoint is inside the drawing area.

```
> (plot (list
         (arrows
          `((0 0) (2 1) (3 3) (0 0))
          #:arrow-head-size-or-scale '(= 20)
          #:arrow-head-angle 0.2
          #:color 6 #:label "a+b+c=0")
         (arrows
          `(((2 0) (0 1)) ((3 0) (-1 1)))
          #:arrow-head-size-or-scale 0.2
          #:color 2 #:label "d")))
```




Added in version 7.9 of package `plot-gui-lib`.

```
(hrule  y
        [x-min
         x-max
         #:color color
         #:width width
         #:style style
         #:alpha alpha
         #:label label]) → renderer2d?
  y : real?
  x-min : (or/c rational? #f) = #f
  x-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? pict? #f) = #f
```

Draws a horizontal line at $y$. By default, the line spans the entire plot area width.

Changed in version 7.9 of package `plot-gui-lib`: Added support for pictures for #:label

```
(vrule  x
        [y-min
         y-max
         #:color color
         #:width width
         #:style style
         #:alpha alpha
         #:label label]) → renderer2d?
  x : real?
  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? pict? #f) = #f
```

Draws a vertical line at $x$. By default, the line spans the entire plot area height.

Changed in version 7.9 of package `plot-gui-lib`: Added support for pictures for #:label

## 3.4 2D Interval Renderers

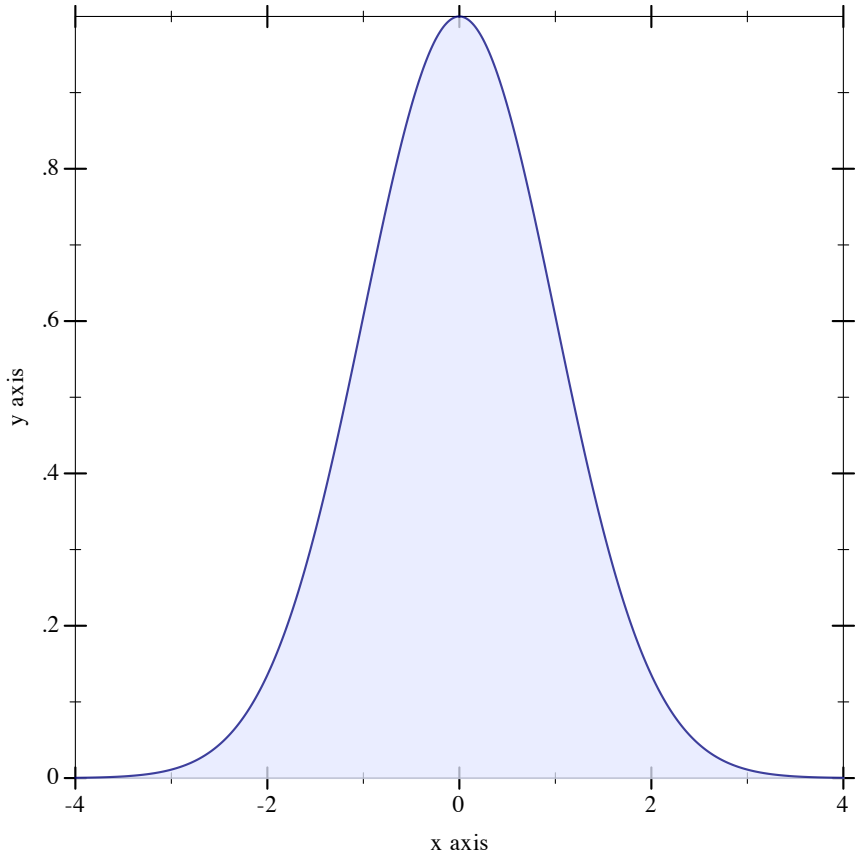
These renderers each correspond with a line renderer, and graph the area between two lines.

```
(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? pict? #f) = #f
```

Corresponds with `function`.

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

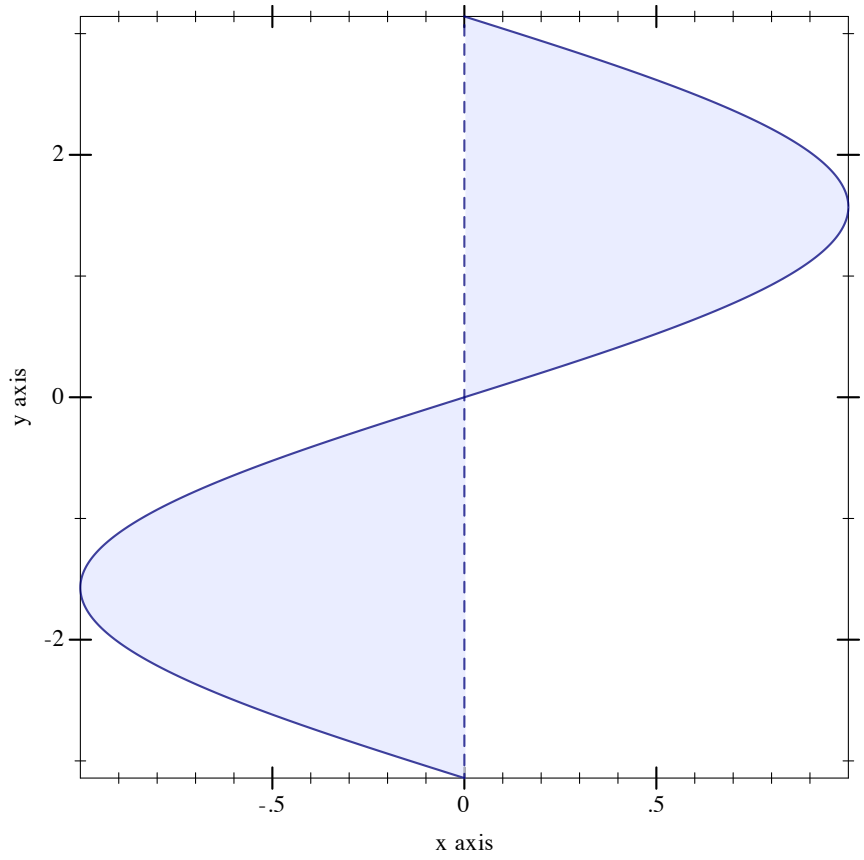



Changed in version 7.9 of package `plot-gui-lib`: Added support for pictures for #:label

```
(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? pict? #f) = #f
```

Corresponds with `inverse`.

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



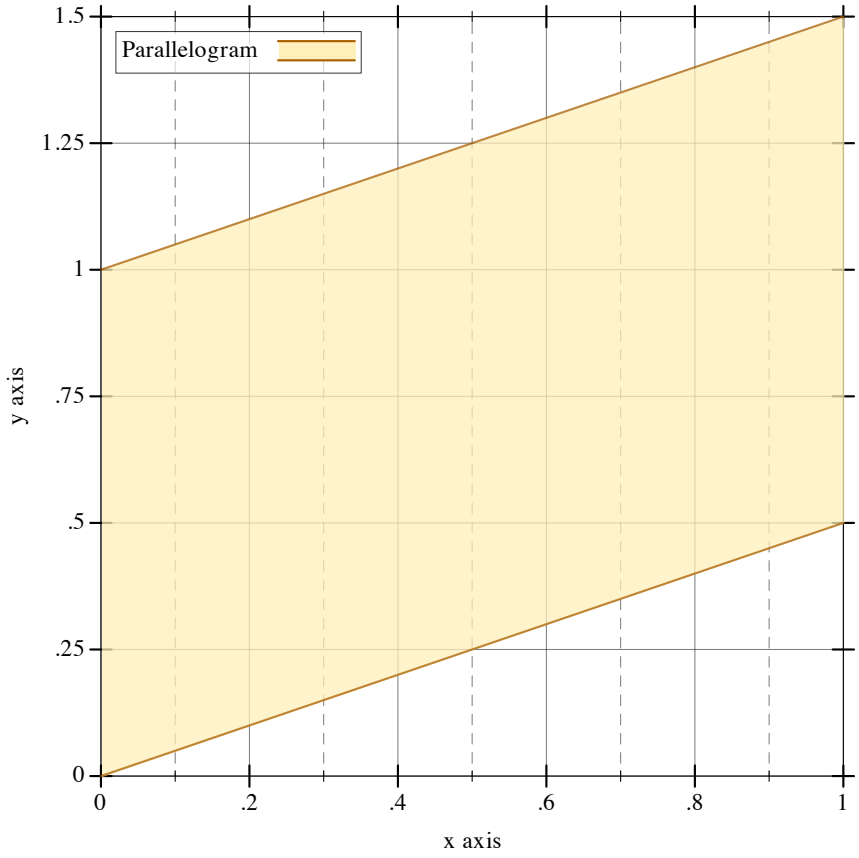

Changed in version 7.9 of package `plot-gui-lib`: Added support for pictures for #:label

```
(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 : (sequence/c (sequence/c #:min-count 2 real?))
v2s : (sequence/c (sequence/c #:min-count 2 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? pict? #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")))
```

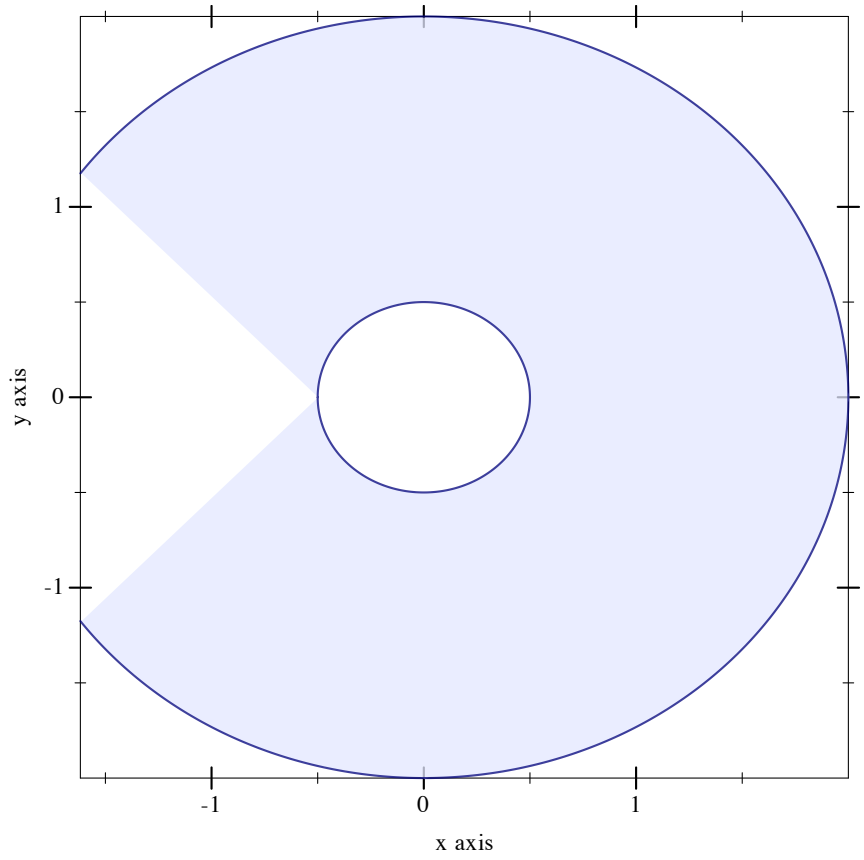



Changed in version 7.9 of package `plot-gui-lib`: Added support for pictures for #:label

```
(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? . -> . (sequence/c real?))
  f2 : (real? . -> . (sequence/c 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? pict? #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))
```

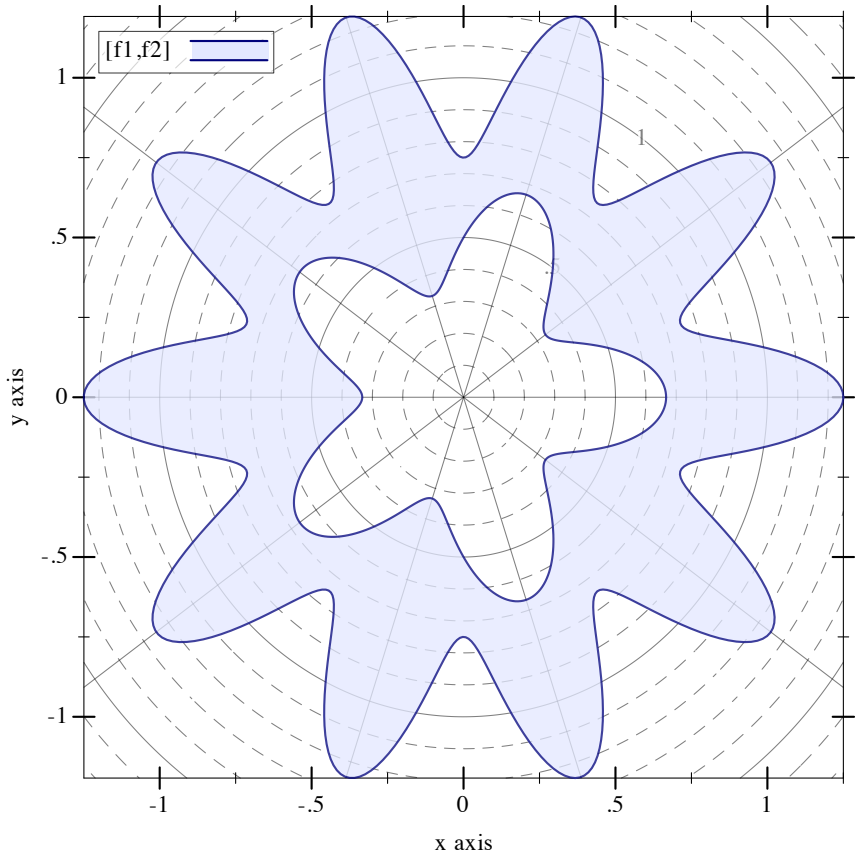



Changed in version 7.9 of package `plot-gui-lib`: Added support for pictures for #:label

```
(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? pict? #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]")))
```




Changed in version 7.9 of package `plot-gui-lib`: Added support for pictures for #:label

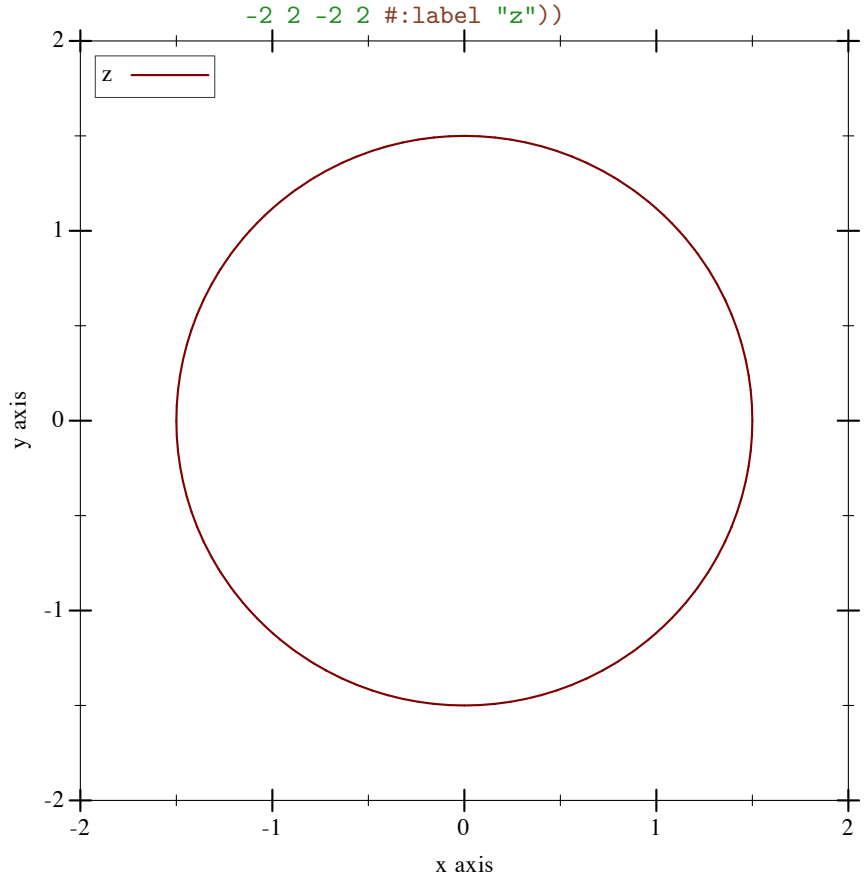
## 3.5 2D Contour (Isoline) Renderers

```
(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? pict? #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"))
```



Changed in version 7.9 of package `plot-gui-lib`: Added support for pictures for #:label

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.

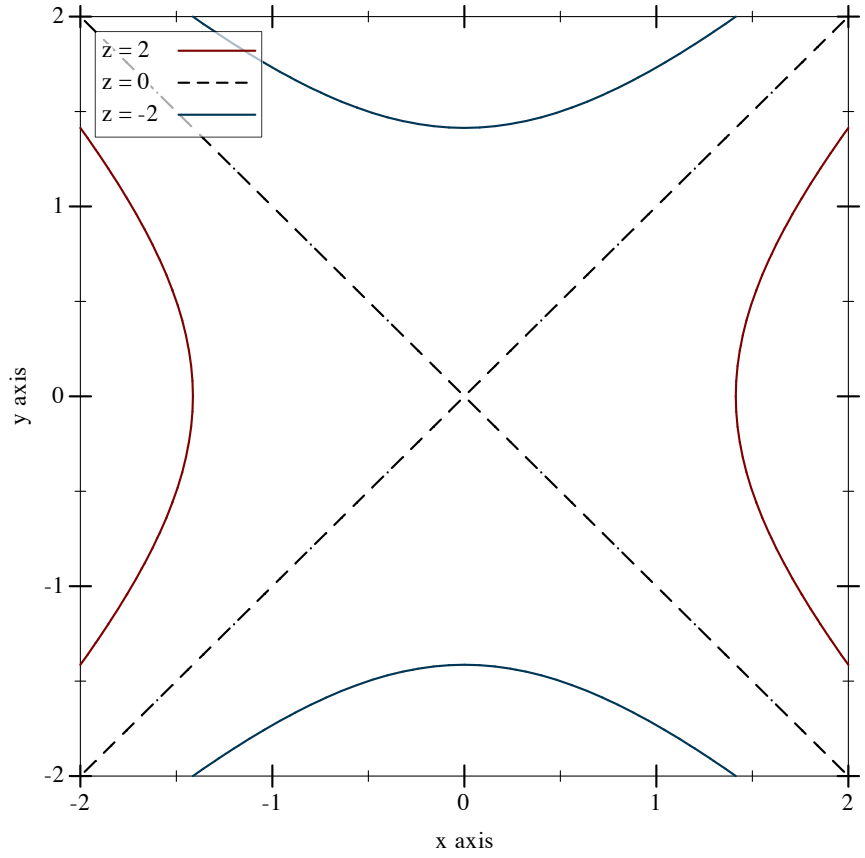
```
(contours f
          [x-min
           x-max
           y-min
           y-max
           #:samples samples
           #:levels levels
           #:colors colors
           #:widths widths
           #:styles styles
           #:alphas alphas
           #:label label])    → renderer2d?
  f : (real? 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? pict? #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:

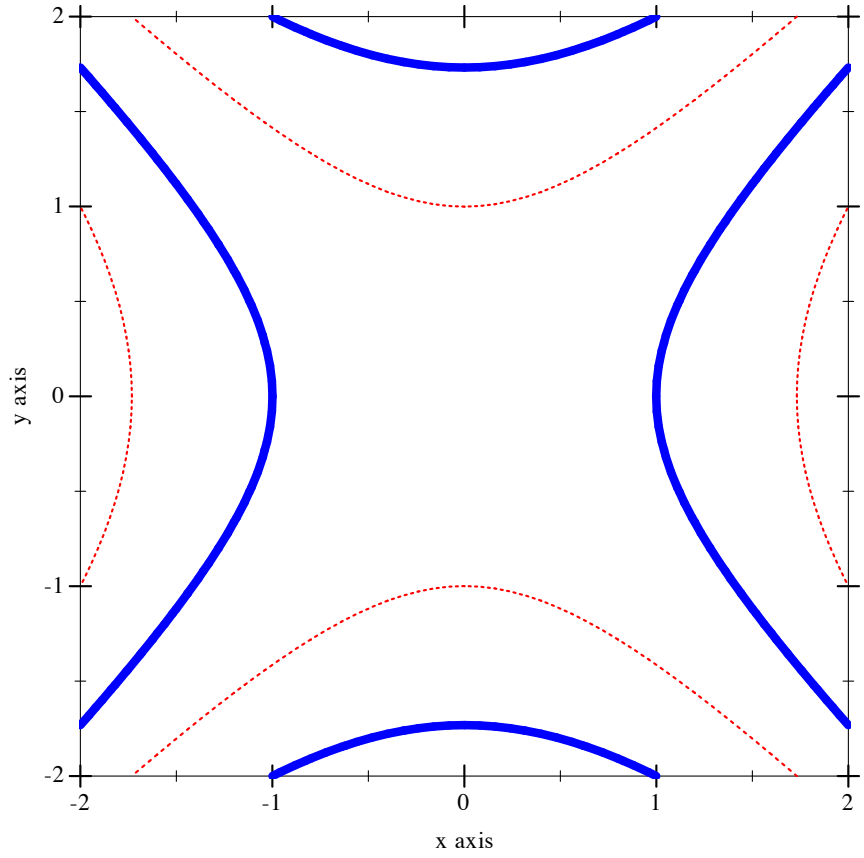
```
> (plot (contours (λ (x y) (- (sqr x) (sqr y)))
                  -2 2 -2 2 #:label "z"))
```




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)))
```




Changed in version 7.9 of package `plot-gui-lib`: Added support for pictures for #:label

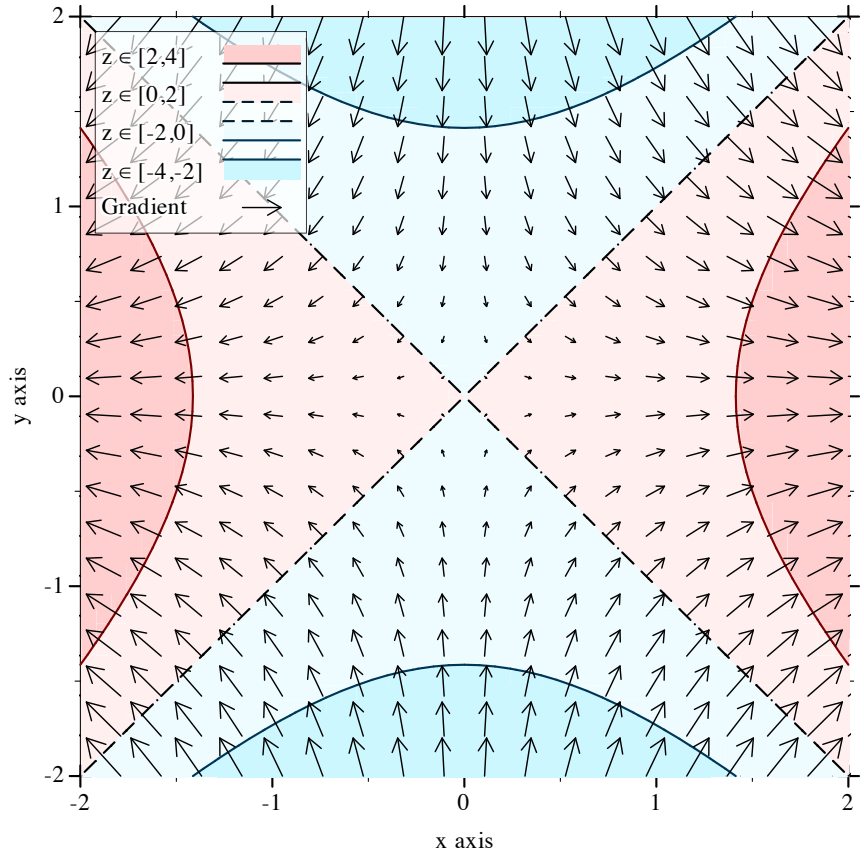
```
(contour-intervals  f
                    [x-min
                     x-max
                     y-min
                     y-max
                     #:samples samples
                     #:levels levels
                     #:colors colors
                     #:styles styles
                     #: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? pict? #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")))
```




Changed in version 7.9 of package `plot-gui-lib`: Added support for pictures for #:label

## 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?
```

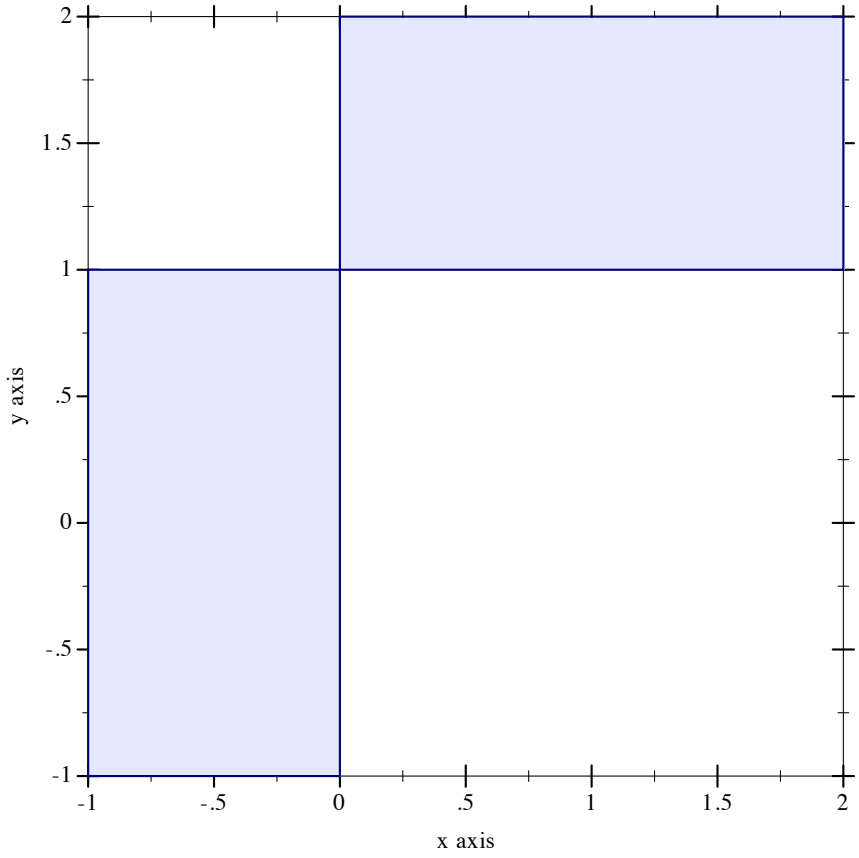
```
rects : (sequence/c (sequence/c #:min-count 2 ivl?))
x-min : (or/c rational? #f) = #f
x-max : (or/c rational? #f) = #f
y-min : (or/c rational? #f) = #f
y-max : (or/c rational? #f) = #f
color : plot-color/c = (rectangle-color)
style : plot-brush-style/c = (rectangle-style)
line-color : plot-color/c = (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? pict? #f) = #f
```

Returns a renderer that draws rectangles.

The rectangles are given as a sequence of sequences of intervals—each inner sequence defines the bounds of a rectangle. Any of the bounds can be `-inf.0` or `+inf.0`, in which case the rectangle extents to the edge of the plot area in the respective direction.

For example,

```
> (plot (rectangles (list (vector (ivl -1 0) (ivl -1 1))
                          (vector (ivl 0 2) (ivl 1 2)))))
```



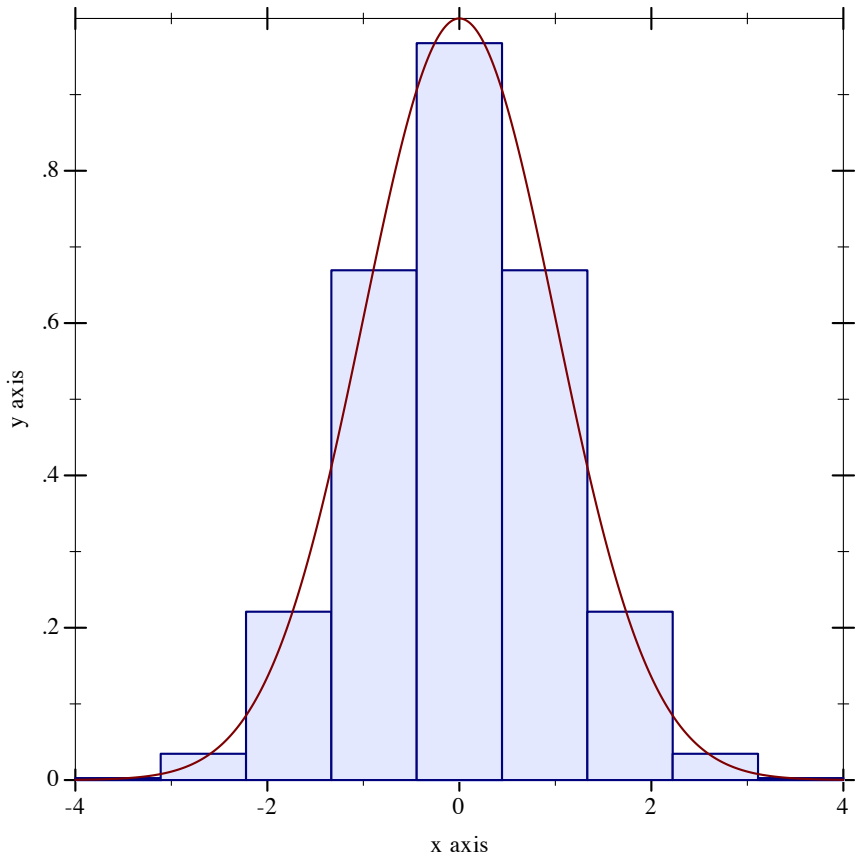

Changed in version 7.9 of package `plot-gui-lib`: Added support for pictures for #:label

```
(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 : (sequence/c 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? pict? #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)))
```



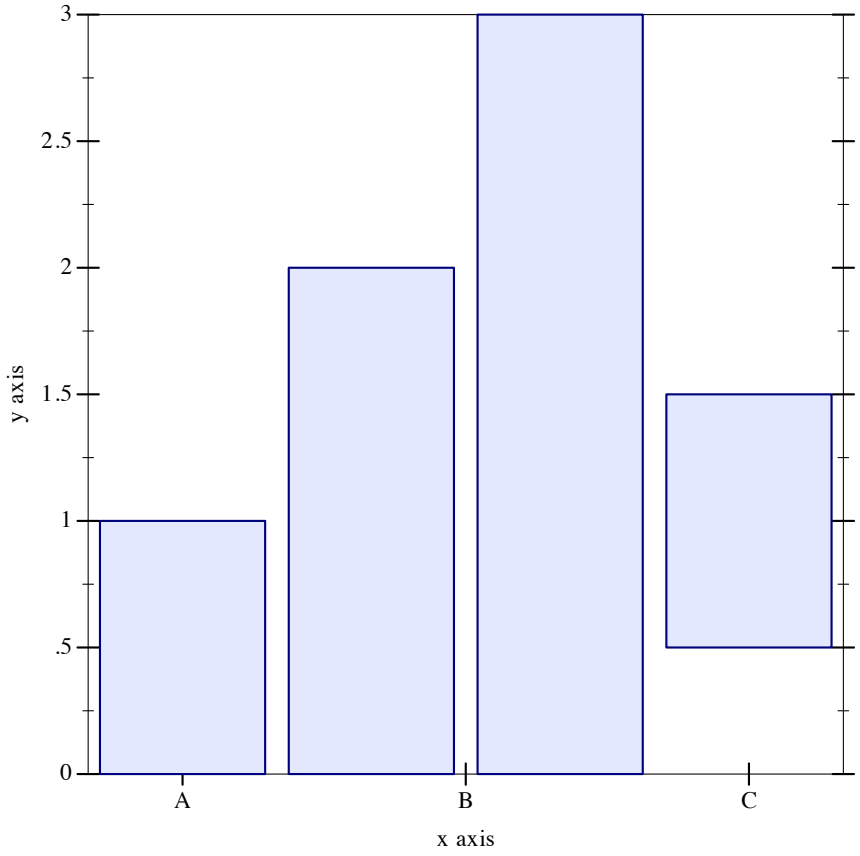

```
(discrete-histogram  cat-vals
                     [#:x-min x-min
                      #:x-max x-max
                      #:y-min y-min
                      #:y-max y-max
                      #:gap gap
                      #:skip skip
                      #:invert? invert?
                      #: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 : (sequence/c (or/c (vector/c any/c (or/c real? ivl? #f))
                             (list/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? pict? #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)))))
```

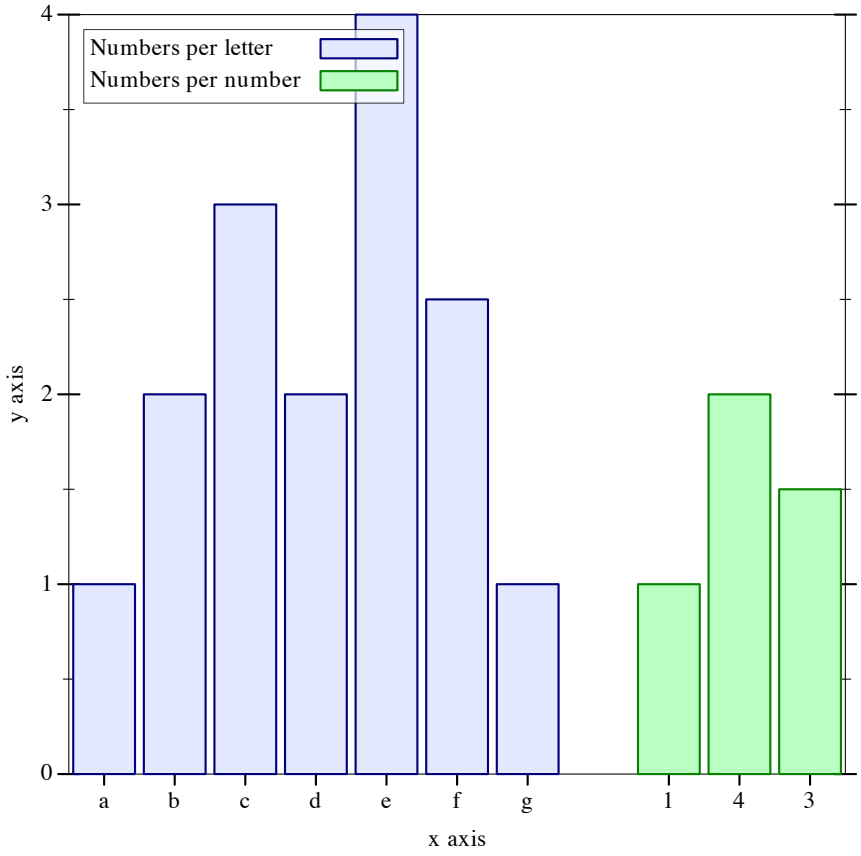



Use `#:invert? #t` to draw horizontal bars. See `stacked-histogram` for an example.

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,

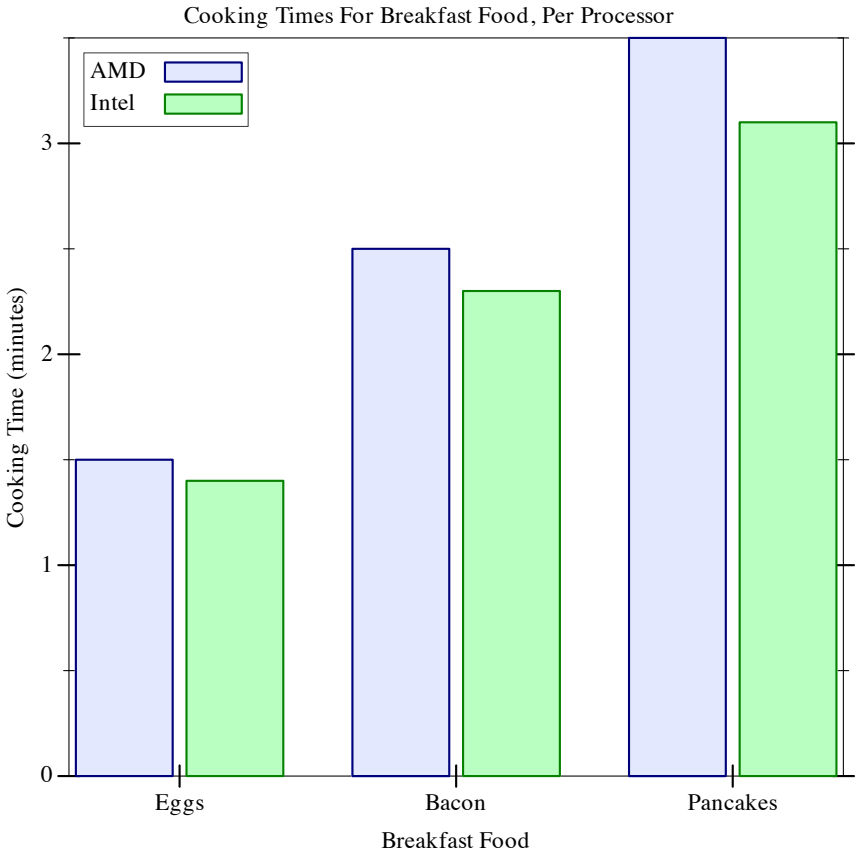
```
> (plot (list (discrete-histogram (list #(a 1) #(b 2) #(c 3) #(d 2)
                                        #(e 4) #(f 2.5) #(g 1))
                                  #:label "Numbers per letter")
              (discrete-histogram (list #(1 1) #(4 2) #(3 1.5))
                                  #:x-min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

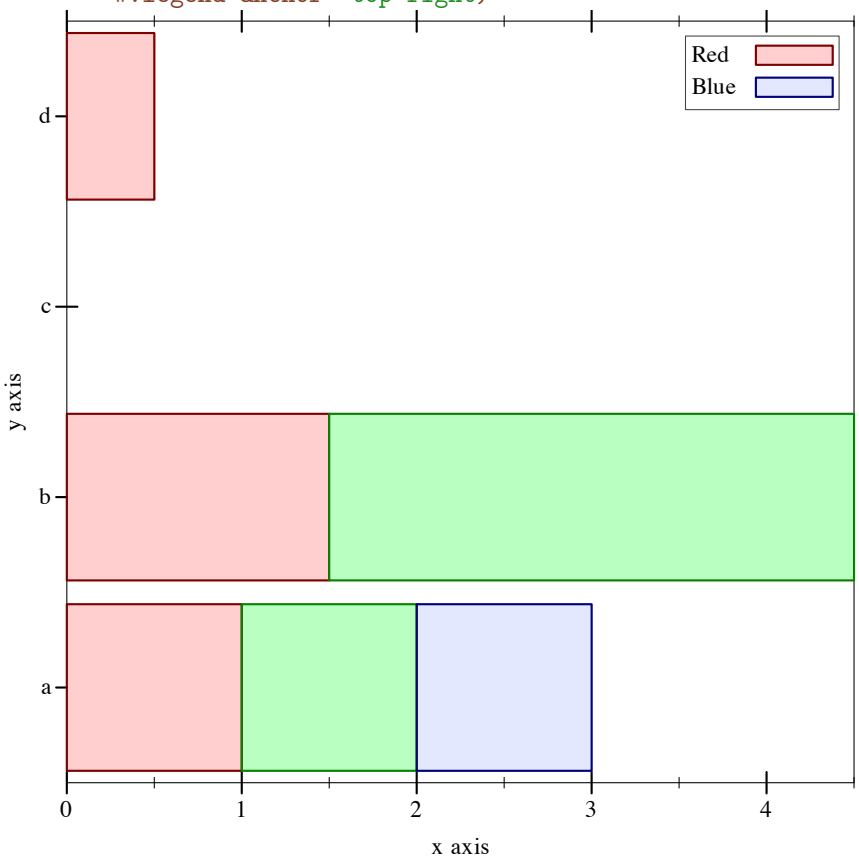
Changed in version 7.9 of package `plot-gui-lib`: Added support for pictures for #:label

```
(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 : (sequence/c (or/c (vector/c any/c (sequence/c real?))
                               (list/c any/c (sequence/c 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 Violin and Box Plot Renderers

```
(violin  vs
         [#:x x
          #:width width
          #:bandwidth bandwidth
          #:invert? invert?
          #:label label
          #:add-ticks? add-ticks?
          #:far-ticks? far-ticks?
          #:y-min y-min
          #:y-max y-max
          #:samples samples
          #:color color
          #:style style
          #:line-color line1-color
          #:line-width line1-width
          #:line-style line1-style
          #:alpha alpha])           → renderer2d?
  vs : (sequence/c real?)
  x : real? = 0
  width : (>=/c 0) = 1
  bandwidth : (or/c real? #f) = #f
  invert? : boolean? = #f
  label : (or/c string? pict? #f) = #f
  add-ticks? : boolean? = #t
  far-ticks? : boolean? = #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)
  alpha : (real-in 0 1) = (interval-alpha)
```

Draws a violin plot from the list of real values *vs*. The plot is centered at *x* and the *width* parameter is used as a scaling factor to control the width of the violin.

The default kernel density bandwidth is determined by `silverman-bandwidth`.

When *invert?* is `#f`, the violin plot is drawn vertically, when it is `#t`, the x and y coordinates are inverted, and the violin is drawn horizontally.

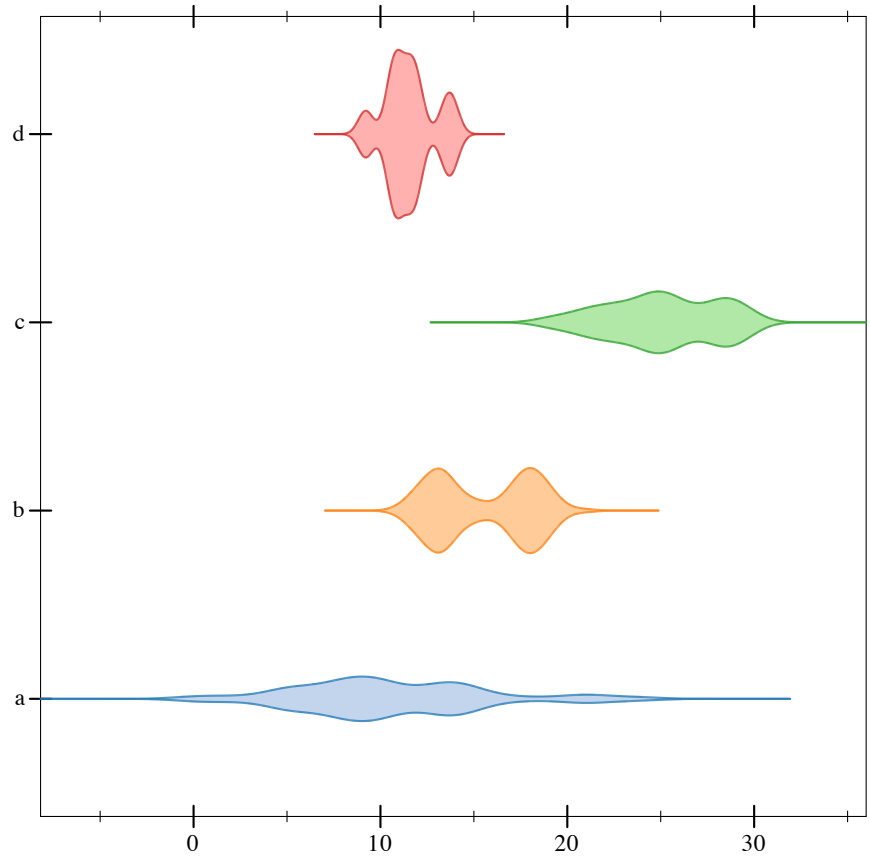
*label* defines the plot label, it is the value shown in the plot legend as well as on the X axis under the violin plot (or Y axis if the plot is inverted). The label is shown on the X axis on only if *add-ticks?* is `#t`, and, if *far-ticks?* is `#t` the label is placed on the far axis.

*y-min* and *y-max* define the vertical range to draw the violin, by default, the entire violin is drawn.

*samples* defines the number of samples used by the function renderer while drawing the violin outline.

See §3.1 “2D Renderer Function Arguments” for the meaning of the other arguments.

```
> (parameterize ([plot-pen-color-map 'tab20]
                 [plot-brush-color-map 'tab20]
                 [plot-x-label #f]
                 [plot-y-label #f])
    (define (rnorm sample-count mean stddev)
      (sample (normal-dist mean stddev) sample-count))
    (define a (rnorm 50 10 5))
    (define b (append (rnorm 50 13 1) (rnorm 50 18 1)))
    (define c (rnorm 20 25 4))
    (define d (rnorm 10 12 2))
    (plot
     (for/list ([data (list a b c d)]
                [label (list "a" "b" "c" "d")]
                [index (in-naturals)])
       (violin data
               #:label label
               #:invert? #t
               #:x index
               #:width 5/4
               #:color (+ (* index 2) 1)
               #:line-color (* index 2)))
     #:legend-anchor 'no-legend))
```




Added in version 8.5 of package `plot-gui-lib`.

```
(box-and-whisker  vs
                  [#:weights ws
                   #:x x
                   #:width width
                   #:iqr-scale iqr-scale
                   #:invert? invert?
                   #:label label
                   #:add-ticks? add-ticks?
                   #:far-ticks? far-ticks?
                   #:box-color box-color
                   #:box-style box-style
                   #:box-line-color box-line-color
                   #:box-line-width box-line-width
                   #:box-line-style box-line-style
                   #:box-alpha box-alpha
                   #:show-outliers? show-outliers?
                   #:outlier-color outlier-color
                   #:outlier-sym outlier-sym
                   #:outlier-fill-color outlier-fill-color
                   #:outlier-size outlier-size
                   #:outlier-line-width outlier-line-width
                   #:outlier-alpha outlier-alpha
                   #:show-whiskers? show-whiskers?
                   #:whisker-color whisker-color
                   #:whisker-width whisker-width
                   #:whisker-style whisker-style
                   #:whisker-alpha whisker-alpha
                   #:show-median? show-median?
                   #:median-color median-color
                   #:median-width median-width
                   #:median-style median-style
                   #:median-alpha median-alpha])
 → renderer2d?
  vs : (sequence/c real?)
  ws : (sequence/c real?) = #f
  x : real? = 0
  width : (>=/c 0) = 1
  iqr-scale : (>=/c 0) = 1.5
  invert? : boolean? = #f
  label : (or/c string? pict? #f) = #f
  add-ticks? : boolean? = #t
  far-ticks? : boolean? = #f
  box-color : plot-color/c = (rectangle-color)
  box-style : plot-brush-style/c = (rectangle-style)
  box-line-color : plot-color/c = (rectangle-line-color)
  box-line-width : (>=/c 0) = (rectangle-line-width)
```

```
box-line-style : plot-pen-style/c = (rectangle-line-style)
box-alpha : (real-in 0 1) = (rectangle-alpha)
show-outliers? : boolean? = #t
outlier-color : plot-color/c = (point-color)
outlier-sym : point-sym/c = (point-sym)
outlier-fill-color : (or/c plot-color/c 'auto) = 'auto
outlier-size : (>=/c 0) = (point-size)
outlier-line-width : (>=/c 0) = (point-line-width)
outlier-alpha : (real-in 0 1) = (point-alpha)
show-whiskers? : boolean? = #t
whisker-color : plot-color/c = (line-color)
whisker-width : (>=/c 0) = (line-width)
whisker-style : plot-pen-style/c = (line-style)
whisker-alpha : (real-in 0 1) = (line-alpha)
show-median? : boolean? = #t
median-color : plot-color/c = (line-color)
median-width : (>=/c 0) = (line-width)
median-style : plot-pen-style/c = (line-style)
median-alpha : (real-in 0 1) = (line-alpha)
```

Draws a box and whisker plot from the list of real values *vs*, possibly weighted by the values in *ws*. The plot is centered at *x* and the *width* parameter is used as a scaling factor to control the width of the box.

The *iqr-scale* controls the scaling factor for the inter-quantile range, which decides how far the whiskers extent and which points are considered outliers.

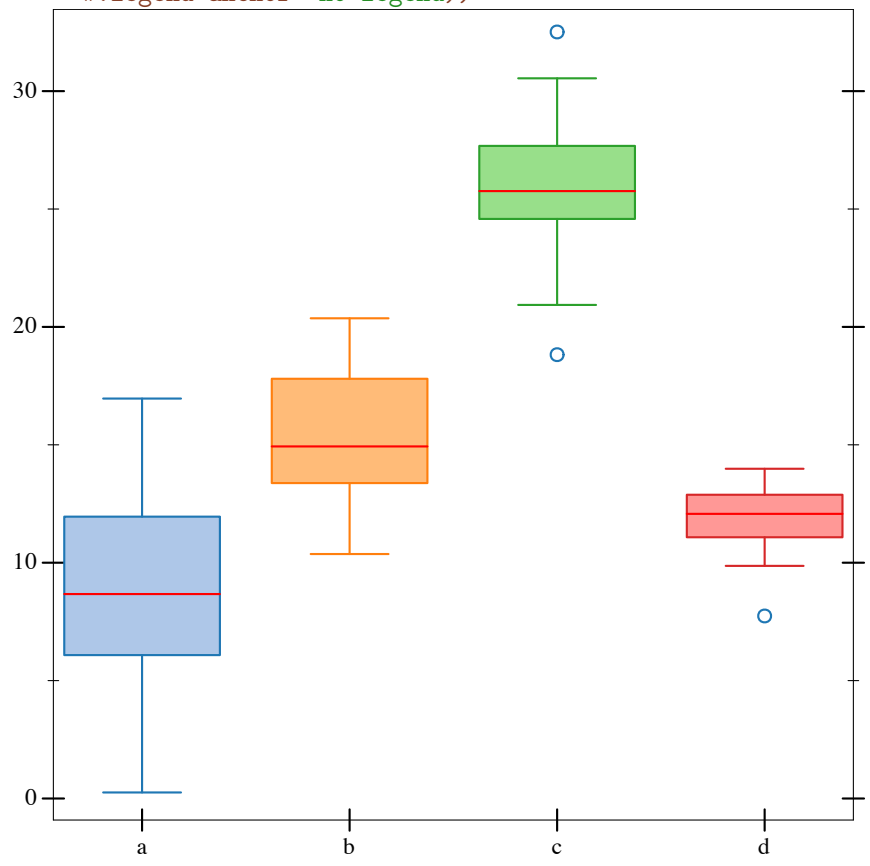
When *invert?* is `#f`, the box plot is drawn vertically, when it is `#t`, the x and y coordinates are inverted, and the box plot is drawn horizontally.

*label* defines the plot label, it is the value shown in the plot legend as well as on the X axis under the box plot (or Y axis if the plot is inverted). The label is shown on the X axis on only if *add-ticks?* is `#t`, and, if *far-ticks?* is `#t` the label is placed on the far axis.

See §3.1 “2D Renderer Function Arguments” for the meaning of the other arguments.

```
> (parameterize ([plot-pen-color-map 'tab20]
                 [plot-brush-color-map 'tab20]
                 [plot-x-label #f]
                 [plot-y-label #f])
    (define (rnorm sample-count mean stddev)
      (sample (normal-dist mean stddev) sample-count))
    (define a (rnorm 50 10 5))
    (define b (append (rnorm 50 13 1) (rnorm 50 18 1)))
    (define c (rnorm 20 25 4))
    (define d (rnorm 10 12 2))
```

```
(plot
 (for/list ([data (list a b c d)]
            [label (list "a" "b" "c" "d")]
            [index (in-naturals)])
   (box-and-whisker data
                    #:label label
                    #:invert? #f
                    #:x index
                    #:width 3/4
                    #:box-color (+ (* index 2) 1)
                    #:box-line-color (* index 2)
                    #:whisker-color (* index 2)
                    #:median-color "red"))
 #:legend-anchor 'no-legend))
```




Added in version 8.5 of package `plot-gui-lib`.

## 3.8 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?)
```

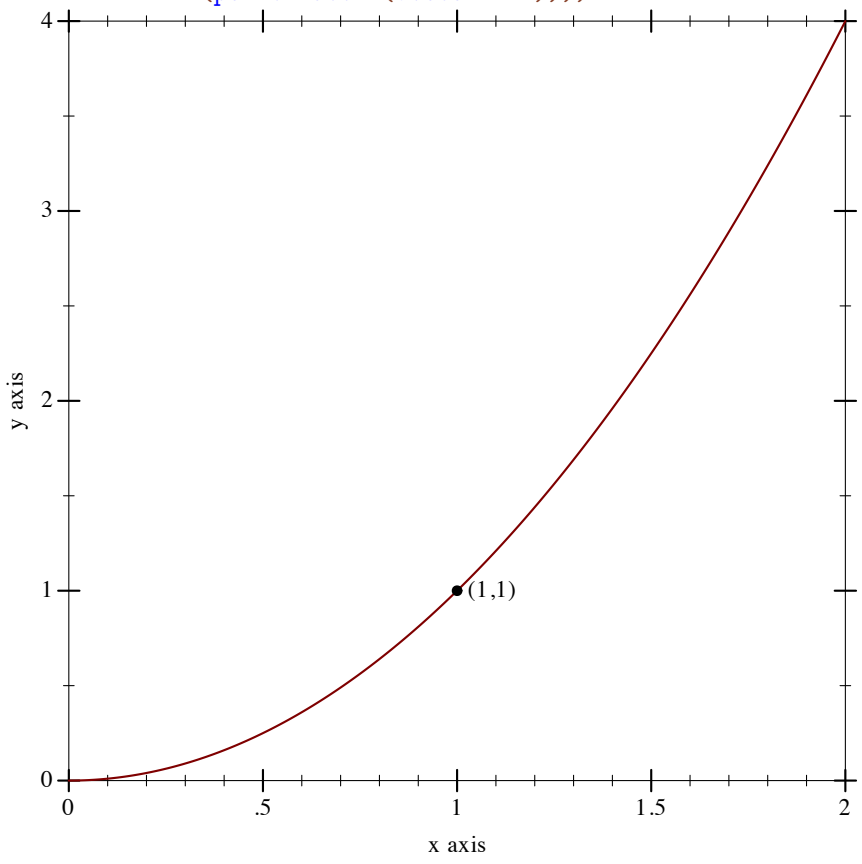
Returns a list containing an `x-tick-lines` renderer and a `y-tick-lines` renderer. See `lines-interval` for an example.

```
(point-label  v
              [label
               #:color color
               #:size size
               #:face face
               #: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 : (sequence/c real?)
  label : (or/c string? #f) = #f
  color : plot-color/c = (plot-foreground)
  size : (>=/c 0) = (plot-font-size)
  face : (or/c string? #f) = (plot-font-face)
  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))))
```

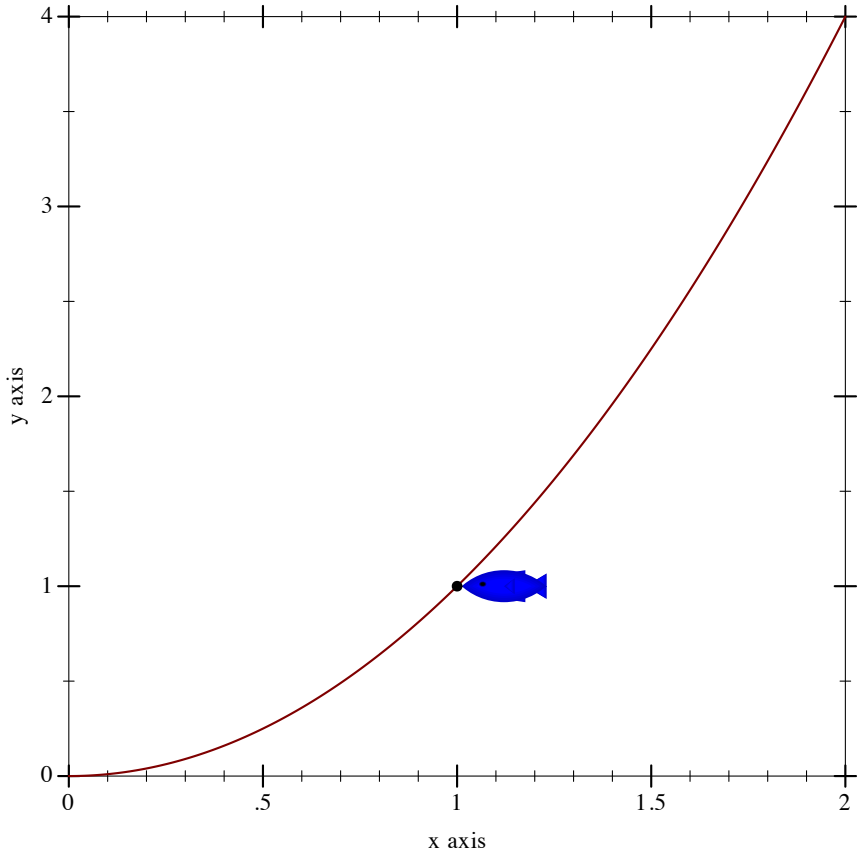



The remaining labeled-point functions are defined in terms of this one.

```
(point-pict  v
             pict
            [#:anchor anchor
             #: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 : (sequence/c real?)
  pict : pict?
  anchor : anchor/c = (label-anchor)
  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 point with a pict as the label.

```
> (require pict)
> (plot (list (function sqr 0 2)
              (point-pict (vector 1 1) (standard-fish 40 15))))
```



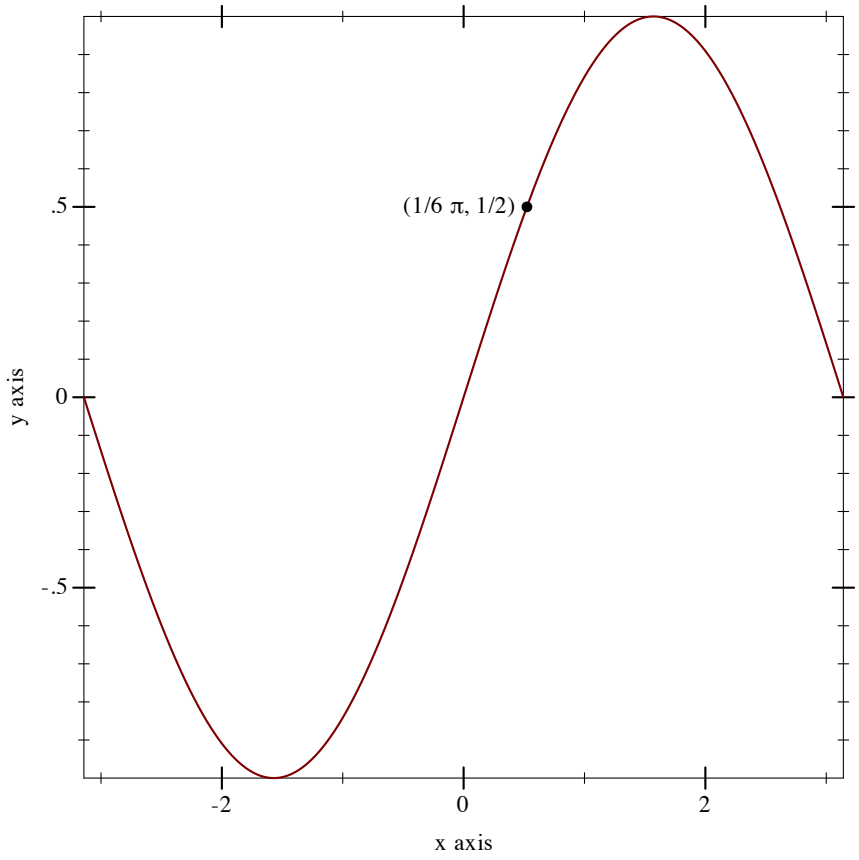

The remaining labeled-pict functions are defined in terms of this one.

```
(function-label  f
                 x
                [label
                 #:color color
                 #:size size
                 #:face face
                 #: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)
face : (or/c string? #f) = (plot-font-face)
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)))
```




```
(function-pict  f
                x
                pict
               [#:anchor anchor
                #: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?
  pict : pict?
  anchor : anchor/c = (label-anchor)
  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 point with a pict as the label on a function's graph.

```
(inverse-label  f
                y
               [label
                #:color color
                #:size size
                #:face face
                #: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)
  face : (or/c string? #f) = (plot-font-face)
  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.

```
(inverse-pict  f
               y
               pict
              [#:anchor anchor
               #: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?
  pict : pict?
  anchor : anchor/c = (label-anchor)
  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 point with a pict as the label on a function's inverted graph.

```
(parametric-label  f
                   t
                  [label
                   #:color color
                   #:size size
                   #:face face
                   #: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? . -> . (sequence/c real?))
  t : real?
  label : (or/c string? #f) = #f
  color : plot-color/c = (plot-foreground)
  size : (>=/c 0) = (plot-font-size)
  face : (or/c string? #f) = (plot-font-face)
```

```
  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.

```
(parametric-pict  f
                  t
                  pict
                 [#:anchor anchor
                  #: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? . -> . (sequence/c real?))
  t : real?
  pict : pict?
  anchor : anchor/c = (label-anchor)
  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 point with a pict as the label on a parametric function's graph.

```
(polar-label  f
              θ
              [label
               #:color color
               #:size size
               #:face face
               #: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)
  face : (or/c string? #f) = (plot-font-face)
  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.

```
(polar-pict  f
             θ
             pict
            [#:anchor anchor
             #: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?
  pict : pict?
```

```
anchor : anchor/c = (label-anchor)
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 point with a pict as the label on a polar function's graph.

## 3.9 Interactive Overlays for 2D plots

```
(require plot/snip)
```
package: plot-gui-lib

A plot snip% object returned by plot-snip can be set up to provide interactive overlays. This feature can be used, for example, to show the current value of the plot function at the mouse cursor.

If the code below is evaluated in DrRacket, the resulting plot will show a vertical line tracking the mouse and the current plot position is shown on a label. This is achieved by adding a mouse callback to the plot snip returned by plot-snip. When the mouse callback is invoked, it will add a vrule at the current X position and a point-label at the current value of the plotted function.

```
(require plot)
(define snip (plot-snip (function sin) #:x-min -5 #:x-max 5))
(define (mouse-callback snip event x y)
  (if (and x y)
      (send snip set-overlay-renderers
            (list (vrule x)
                  (point-label (vector x (sin x)))))
      (send snip set-overlay-renderers #f)))
(send snip set-mouse-event-callback mouse-callback)
snip
```

Here are a few hints for adding common interactive elements to racket plots:

- The hrule and vrule renderers can be used to draw horizontal and vertical lines that track the mouse position
- The rectangles renderer can be used to highlight a region on the plot. For example, to highlight a vertical region between *xmin* and *xmax*, you can use:

```
(rectangles (list (vector (ivl xmin xmax) (ivl -inf.0 +inf.0)))
            #:alpha 0.2)
```

- A point-label renderer can be used to add a point with a string label to the plot. To add only the label, use 'none as the value for the #:point-sym argument.
- A point-pict renderer can be used to add a point with an attached pict instead of a string label. This can be used to draw fancy labels (for example with rounded corners), or any other type of graphics element.
- A points renderer can be used to mark specific locations on the plot, without specifying a label for them

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

An instance of this class is returned by plot-snip.

```
(send a-2d-plot-snip set-mouse-event-
callback callback) → any/c
  callback : (or/c plot-mouse-event-callback/c #f)
```

Set a callback function to be invoked with mouse events from the snip. The callback is invoked with the actual snip object, the mouse-event% and the X, Y position of the mouse in plot coordinates (i.e., the coordinate system used by the renderers in the plot). The X and Y values are #f when the mouse is outside the plot area (for example, when the mouse is over the axis area).

When a callback is installed, the default zoom functionality of the plot snips is disabled. This can be restored by calling set-mouse-event-callback with a #f argument.

```
(send a-2d-plot-snip set-overlay-renderers renderers) → any/c
  renderers : (or/c (treeof renderer2d?) #f)
```

Set a collection of renderers to be drawn on top of the existing plot. This can be any combination of 2D renderers, but it will not be able to modify the axes or the dimensions of the plot area. Only one set of overlay renderers can be installed; calling this method a second time will replace the previous overlays. Specifying #f as the renderers will cause overlays to be disabled.

```
plot-mouse-event-callback/c : contract?
 = (-> (is-a?/c snip%)
       (is-a?/c mouse-event%)
       (or/c real? #f)
       (or/c real? #f)
       any/c)
```

A contract for callback functions passed to set-mouse-event-callback.

# 4 3D Renderers

```
(require plot)
```
package: `plot-gui-lib`

## 4.1 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.

## 4.2 3D Point Renderers

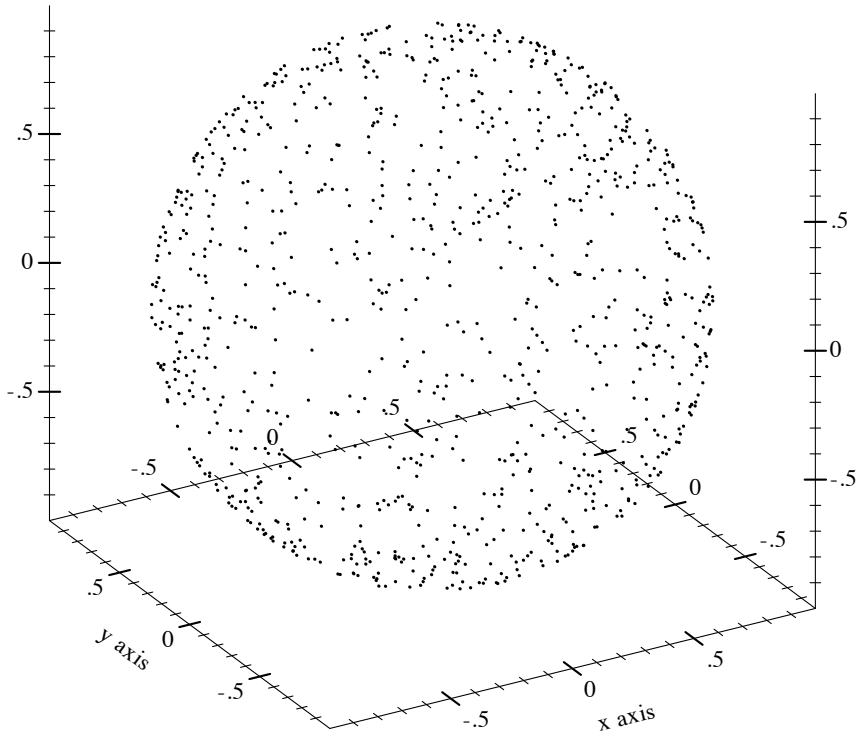
```
(points3d  vs
           [#:x-min x-min
            #:x-max x-max
            #:y-min y-min
            #:y-max y-max
            #:z-min z-min
            #:z-max z-max
            #:sym sym
            #:color color
            #:fill-color fill-color
            #:x-jitter x-jitter
            #:y-jitter y-jitter
            #:z-jitter z-jitter
            #:size size
            #:line-width line-width
            #:alpha alpha
            #:label label])          → renderer3d?
  vs : (sequence/c (sequence/c #:min-count 3 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
x-jitter : (>=/c 0) = (point-x-jitter)
y-jitter : (>=/c 0) = (point-y-jitter)
z-jitter : (>=/c 0) = (point-z-jitter)
size : (>=/c 0) = (point-size)
line-width : (>=/c 0) = (point-line-width)
alpha : (real-in 0 1) = (point-alpha)
label : (or/c string? pict? #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)
```




When `x-jitter`, `y-jitter`, or `z-jitter` is non-zero, each point `p` is translated along the matching axis by a random distance no greater than the given value. Jitter may be applied in either the positive or negative direction, so total spread along e.g. the x-axis is twice `x-jitter`.

Note that adding random noise to data, via jittering or otherwise, is usually a bad idea. See the documentation for `points` for examples where jittering may be appropriate.

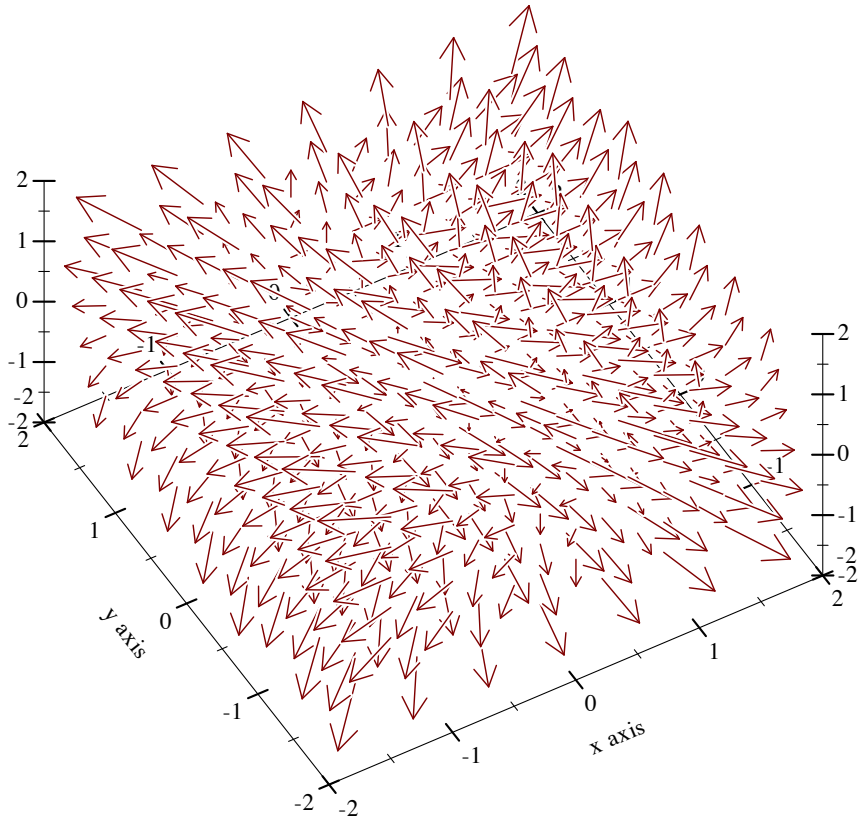
Changed in version 7.9 of package `plot-gui-lib`: Added support for pictures for #:label

```
(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?
  f : (or/c (real? real? real? . -> . (sequence/c real?))
            ((vector/c real? real? real?) . -> . (sequence/c 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? pict? #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))
```




Changed in version 7.9 of package `plot-gui-lib`: Added support for pictures for #:label and controlling the arrowhead

## 4.3 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 : (sequence/c (sequence/c #:min-count 3 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? pict? #f) = #f
```

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

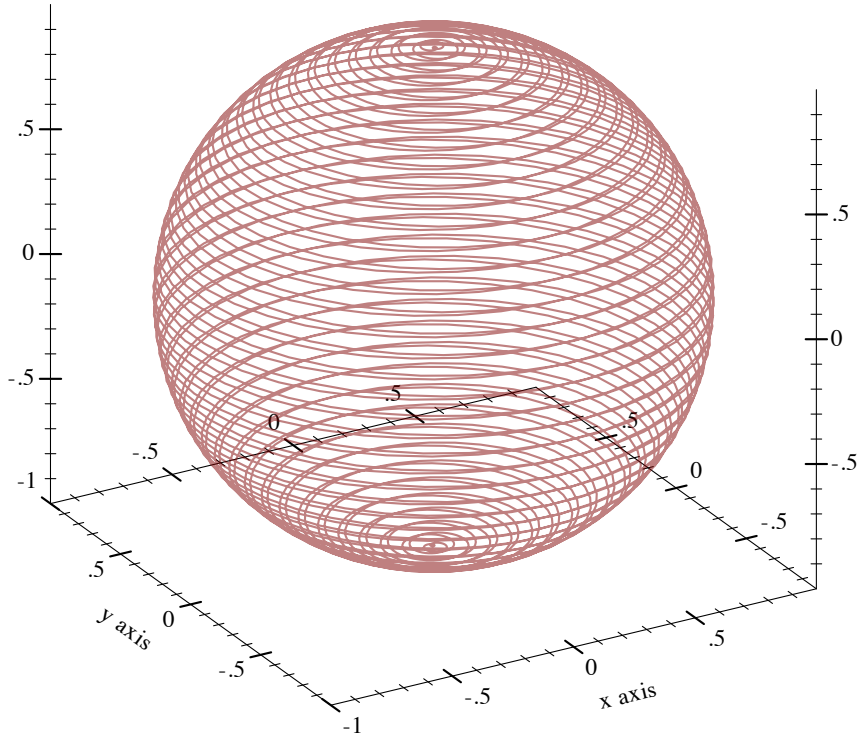
Changed in version 7.9 of package plot-gui-lib: Added support for pictures for #:label

```
(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? . -> . (sequence/c 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? pict? #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)
```

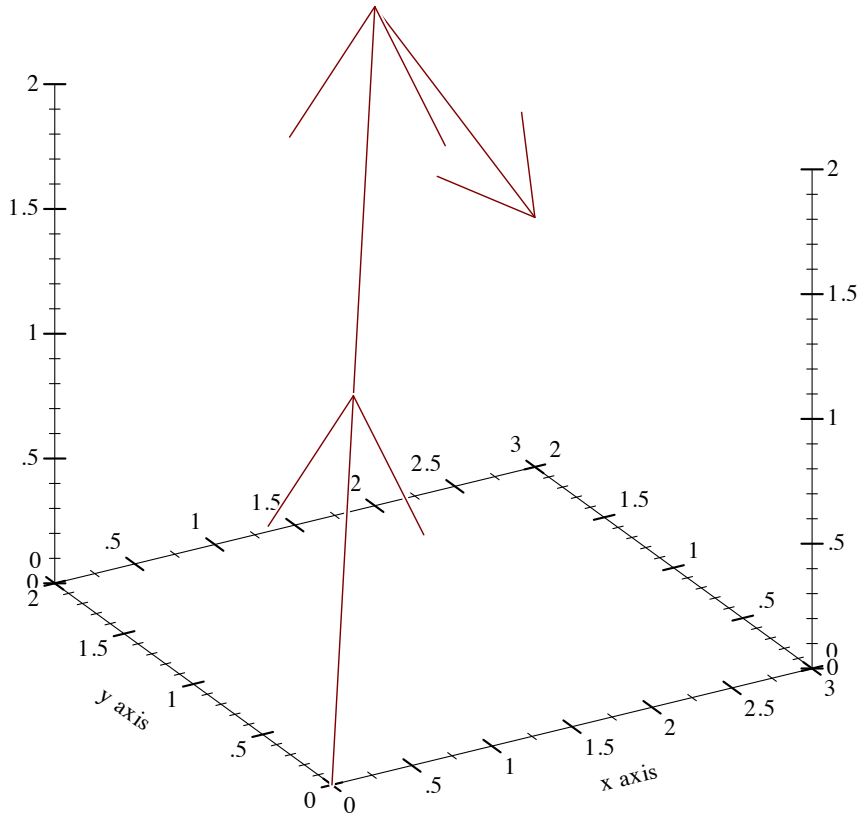



Changed in version 7.9 of package `plot-gui-lib`: Added support for pictures for #:label

```
(arrows3d  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
            #:arrow-head-size-or-scale size
            #:arrow-head-angle angle
            #:label label])                  → renderer3d?
  vs : (or/c (listof (sequence/c #:min-count 3 real?))
             (vectorof (vector/c (sequence/c #:min-count 3 real?)
                                 (sequence/c #:min-count 3 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 = (arrows-color)
  width : (>=/c 0) = (arrows-line-width)
  style : plot-pen-style/c = (arrows-line-style)
  alpha : (real-in 0 1) = (arrows-alpha)
  size : (or/c (list/c '= (>=/c 0)) (>=/c 0))
       = (arrow-head-size-or-scale)
  angle : (>=/c 0) = (arrow-head-angle)
  label : (or/c string? pict? #f) = #f
```

Returns a renderer that draws arrows. The arguments and arrow-head parameters are interpreted identically as in `arrows`.

```
> (plot3d (arrows3d `((0 0 0) (1 1 1) (2 2 2) (3 2 1)))
          #:altitude 25)
```



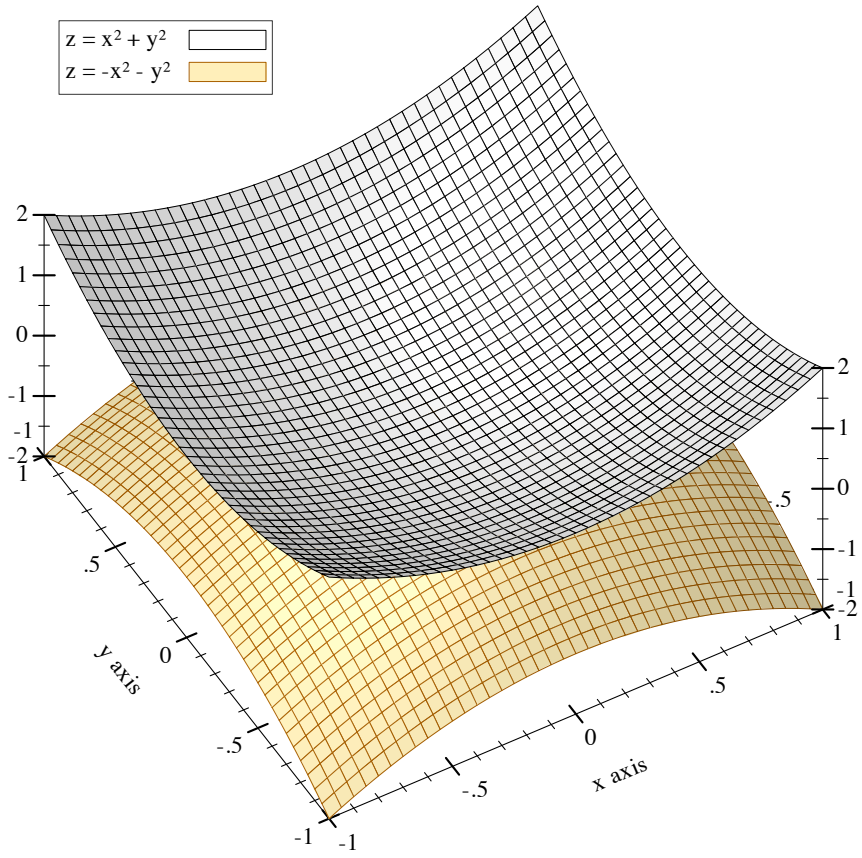

Added in version 7.9 of package `plot-gui-lib`.

## 4.4 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? pict? #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")))
```




Changed in version 7.9 of package `plot-gui-lib`: Added support for pictures for #:label

```
(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?)
```

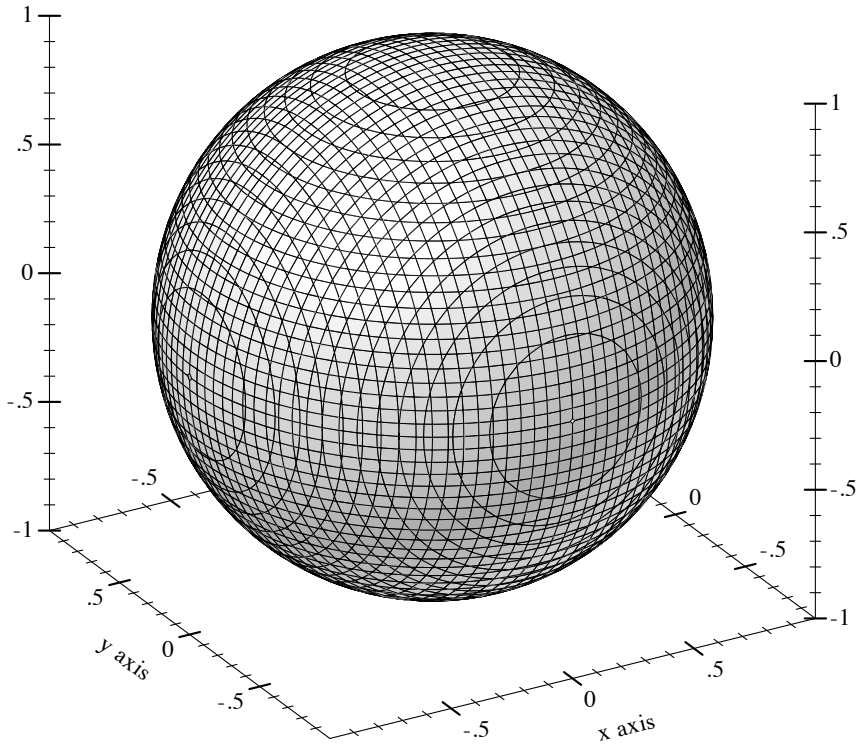
```
x-min : (or/c rational? #f) = #f
x-max : (or/c rational? #f) = #f
y-min : (or/c rational? #f) = #f
y-max : (or/c rational? #f) = #f
z-min : (or/c rational? #f) = #f
z-max : (or/c rational? #f) = #f
samples : (and/c exact-integer? (>=/c 2)) = (plot3d-samples)
color : plot-color/c = (surface-color)
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? pict? #f) = #f
```

Returns a renderer that plots a function from longitude and latitude to radius. `(f θ ϕ)` → `r`

Currently, longitudes($\theta$) range from `0` to `(* 2 pi)`, and latitudes($\phi$) 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:

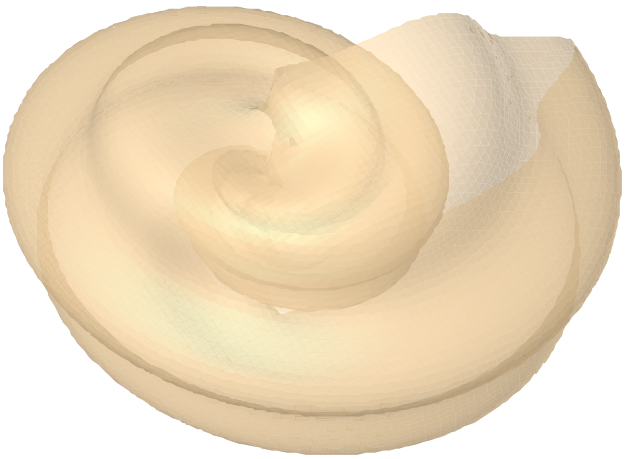
```
> (plot3d (polar3d (λ (θ ϕ) 1)) #:altitude 25)
```




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 ϕ) (+ 1 (/ θ 2 pi) (* 1/8 (sin (* 8 ϕ)))))
    (define (f2 θ ϕ) (+ 0 (/ θ 2 pi) (* 1/8 (sin (* 8 ϕ)))))

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



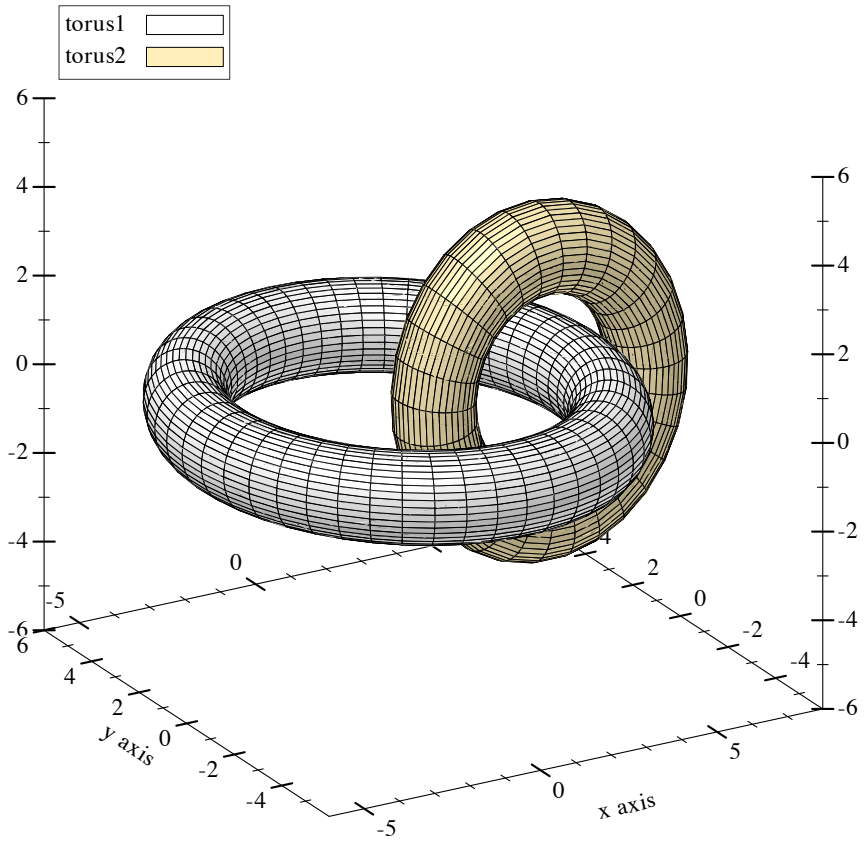
Changed in version 7.9 of package `plot-gui-lib`: Added support for pictures for #:label

```
(parametric-surface3d  f
                       s-min
                       s-max
                       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
                       #:s-samples s-samples
                       #:t-samples t-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? . -> . (sequence/c real?))
  s-min : rational?
  s-max : rational?
  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)) = (plot3d-samples)
  s-samples : (and/c exact-integer? (>=/c 2)) = samples
  t-samples : (and/c exact-integer? (>=/c 2)) = 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? pict? #f) = #f
```

Returns a renderer that plots a two-input, one-output function. `(f s t)` → `'(x y z)`

For example,

```
> (plot3d (list
           (parametric-surface3d
            (λ (θ ϕ)
              (list (* (+ 5 (sin ϕ)) (sin θ))
                    (* (+ 5 (sin ϕ)) (cos θ))
                    (+ 0 (cos ϕ))))
            0 (* 2 pi) #:s-samples 50
            0 (* 2 pi)
            #:label "torus1")
           (parametric-surface3d
            (λ (θ ϕ)
              (list (+ 4 (* (+ 3 (sin ϕ)) (sin θ)))
                    (+ 0 (cos ϕ))
                    (* (+ 3 (sin ϕ)) (cos θ))))
            0 (* 2 pi) #:s-samples 30
            0 (* 2 pi)
            #:color 4
            #:label "torus2"))
          #:z-min -6 #:z-max 6
          #:altitude 22)
```



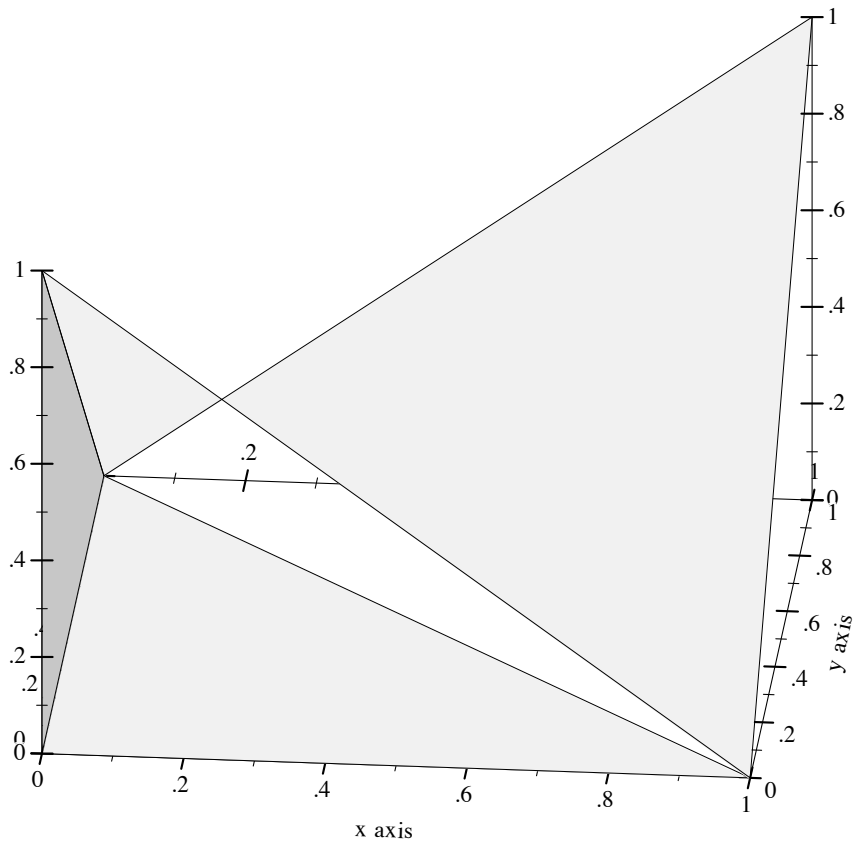

Changed in version 7.9 of package `plot-gui-lib`: Added support for pictures for #:label

```
(polygons3d  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
              #:style style
              #:line-color line-color
              #:line-width line-width
              #:line-style line-style
              #:alpha alpha
              #:label label])          → renderer3d?
  vs : (sequence/c (sequence/c (sequence/c 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 = (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? pict? #f) = #f
```

Returns a renderer that draws polygons. The `parametric-surface3d` function is defined in terms of this one.

```
> (plot3d
   (polygons3d (list (list (list 1 0 0)(list 0 0 1)(list 0 1 0)(list 1 1 1))
                     (list (list 0 0 0)(list 0 0 1)(list 0 1 0))
                     (list (list 1 0 0)(list 0 1 0)(list 0 0 0))))
   #:angle 355
   #:altitude 30)
```




Changed in version 7.9 of package `plot-gui-lib`: Added support for pictures for #:label

## 4.5 3D Contour (Isoline) Renderers

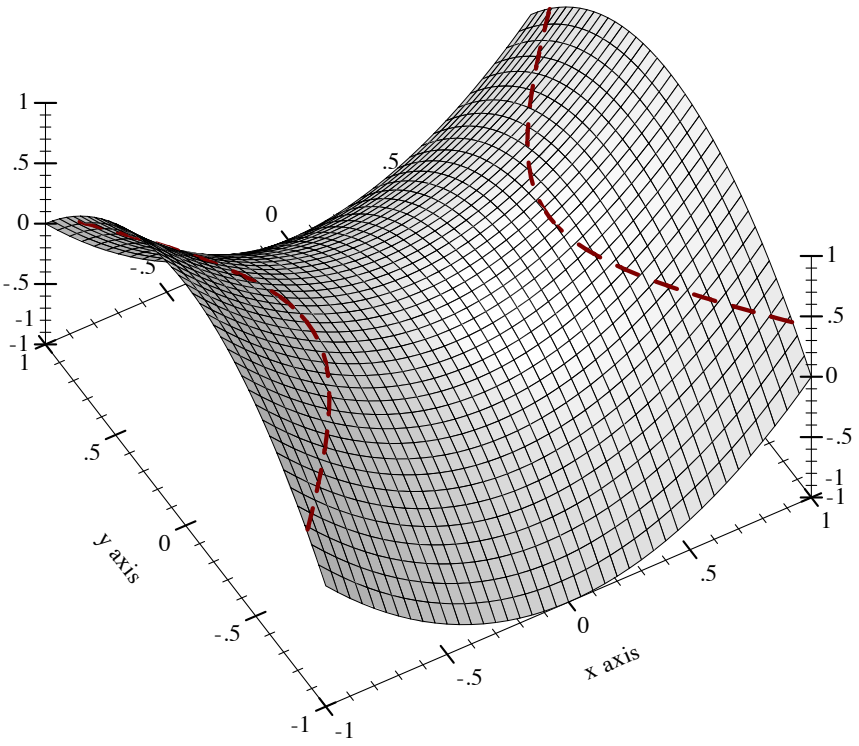
```
(isoline3d  f
            z
           [x-min
            x-max
            y-min
            y-max
            #:z-min z-min
            #:z-max z-max
            #:samples samples
            #:color color
            #:width width
            #:style style
            #:alpha alpha
            #: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? pict? #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)))
```




Changed in version 7.9 of package `plot-gui-lib`: Added support for pictures for #:label

```
(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
```

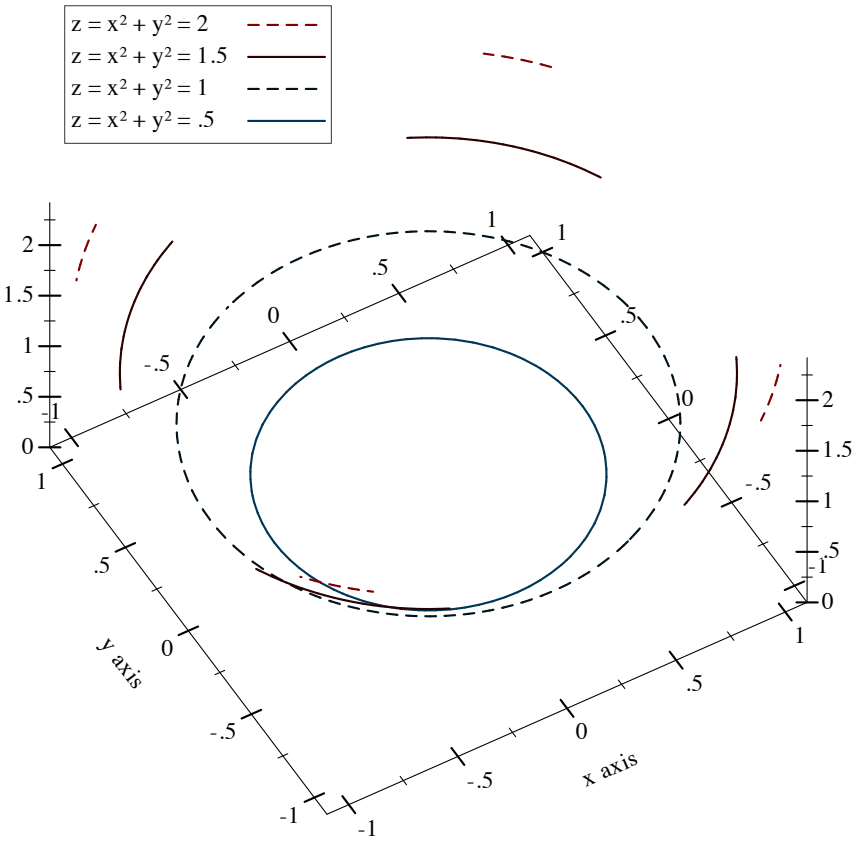
```
x-max : (or/c rational? #f) = #f
y-min : (or/c rational? #f) = #f
y-max : (or/c rational? #f) = #f
z-min : (or/c rational? #f) = #f
z-max : (or/c rational? #f) = #f
samples : (and/c exact-integer? (>=/c 2)) = (plot3d-samples)
levels : (or/c 'auto exact-positive-integer? (listof real?))
       = (contour-levels)
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? pict? #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"))
```




Changed in version 7.9 of package `plot-gui-lib`: Added support for pictures for #:label

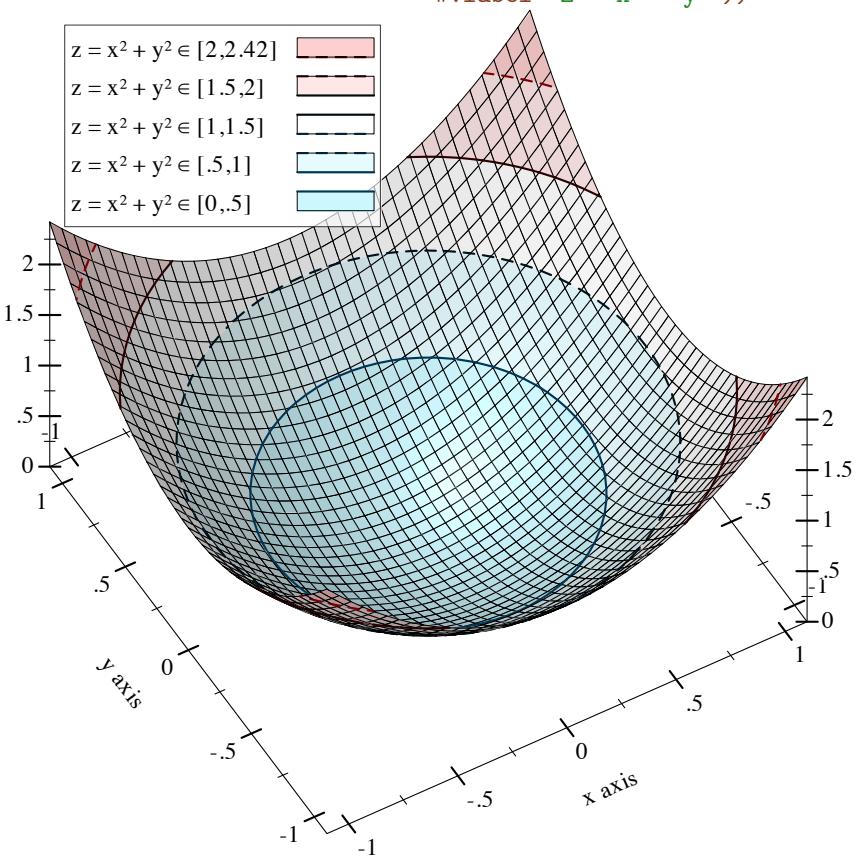
```
(contour-intervals3d  f
                     [x-min
                      x-max
                      y-min
                      y-max
                      #:z-min z-min
                      #:z-max z-max
                      #:samples samples
                      #:levels levels
                      #: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? pict? #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"))
```




Changed in version 7.9 of package `plot-gui-lib`: Added support for pictures for #:label

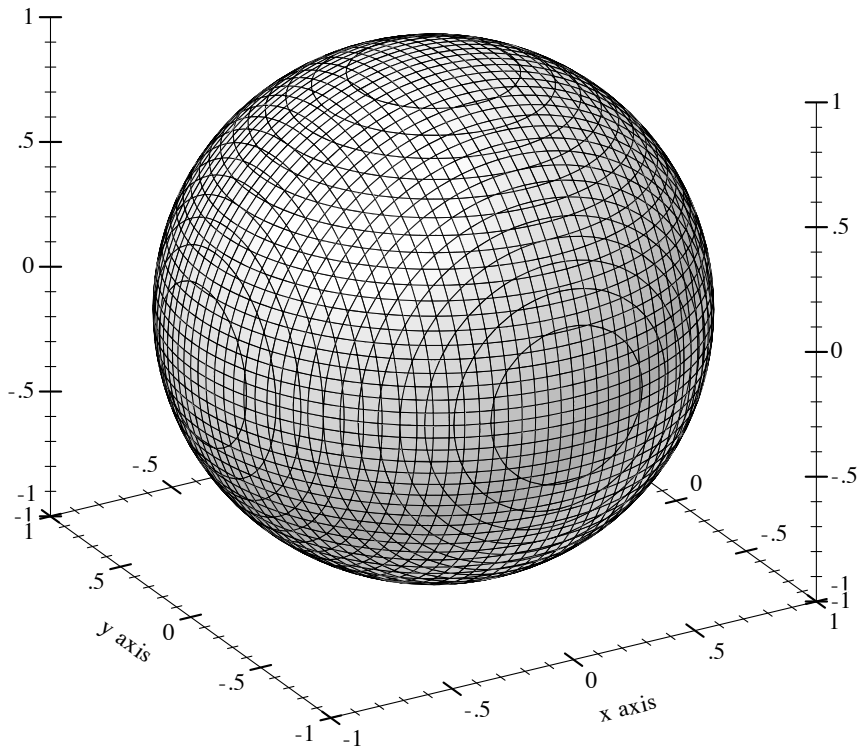
## 4.6 3D Isosurface Renderers

```
(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? pict? #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)
```



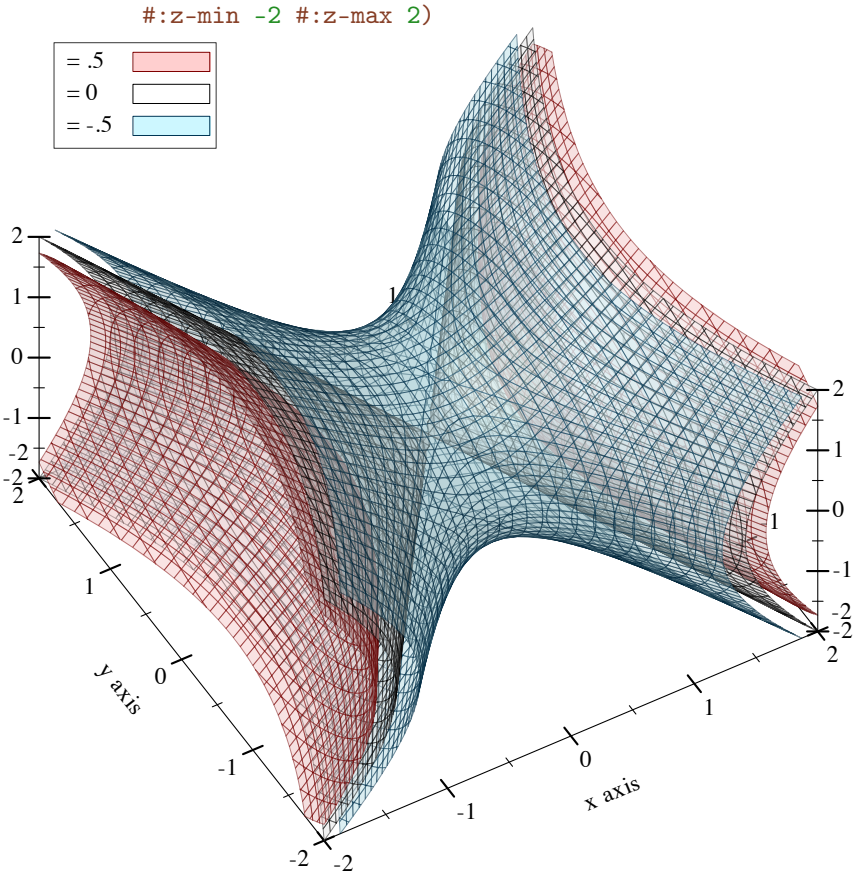

Changed in version 7.9 of package `plot-gui-lib`: Added support for pictures for #:label

```
(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? pict? #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.

Changed in version 7.9 of package `plot-gui-lib`: Added support for pictures for #:label

## 4.7 3D Rectangle Renderers

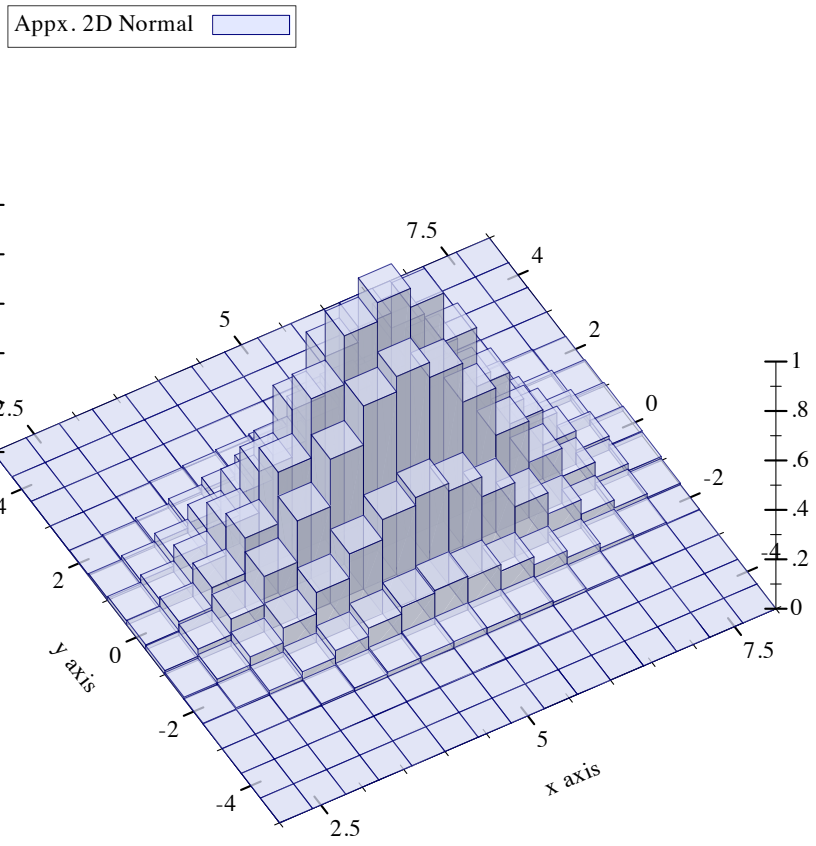
```
(rectangles3d  rects
               [#:x-min x-min
                #:x-max x-max
                #:y-min y-min
                #:y-max y-max
                #:z-min z-min
                #:z-max z-max
                #:color color
                #:style style
                #:line-color line-color
                #:line-width line-width
                #:line-style line-style
                #:alpha alpha
                #:label label])          → renderer3d?
  rects : (sequence/c (sequence/c #:min-count 3 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? pict? #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"))
```




Changed in version 7.9 of package `plot-gui-lib`: Added support for pictures for #:label

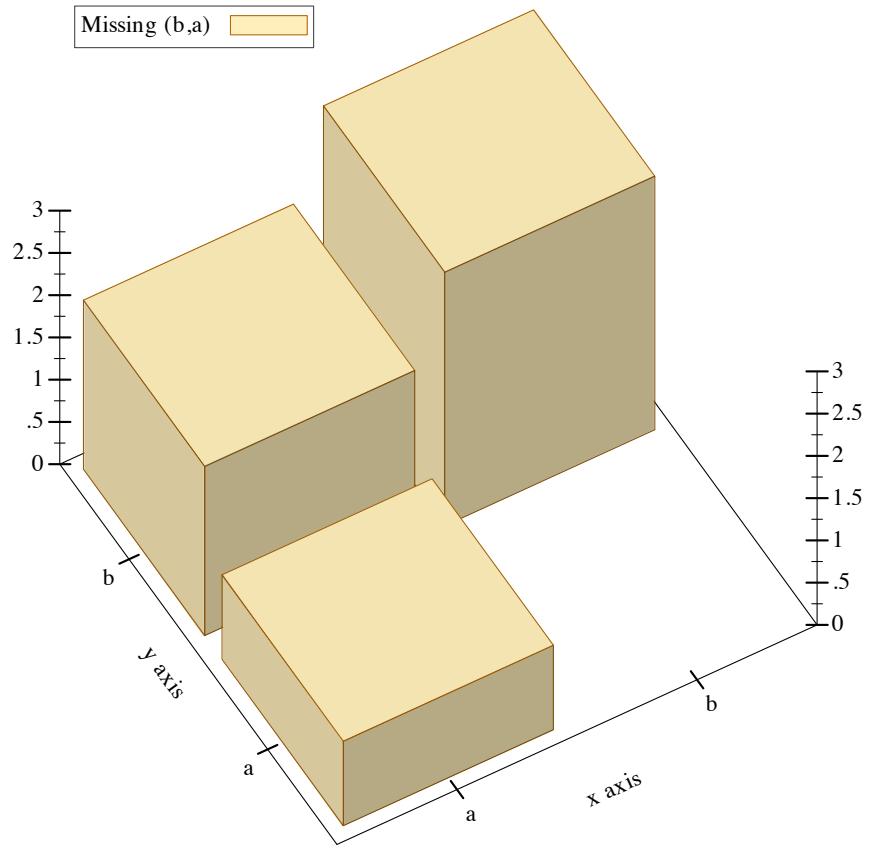
```
(discrete-histogram3d  cat-vals
                       [#:x-min x-min
                        #:x-max x-max
                        #:y-min y-min
                        #:y-max y-max
                        #:z-min z-min
                        #:z-max z-max
                        #:gap gap
                        #: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 : (sequence/c (or/c (vector/c any/c any/c (or/c real? ivl? #f))
                               (list/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? pict? #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))
```



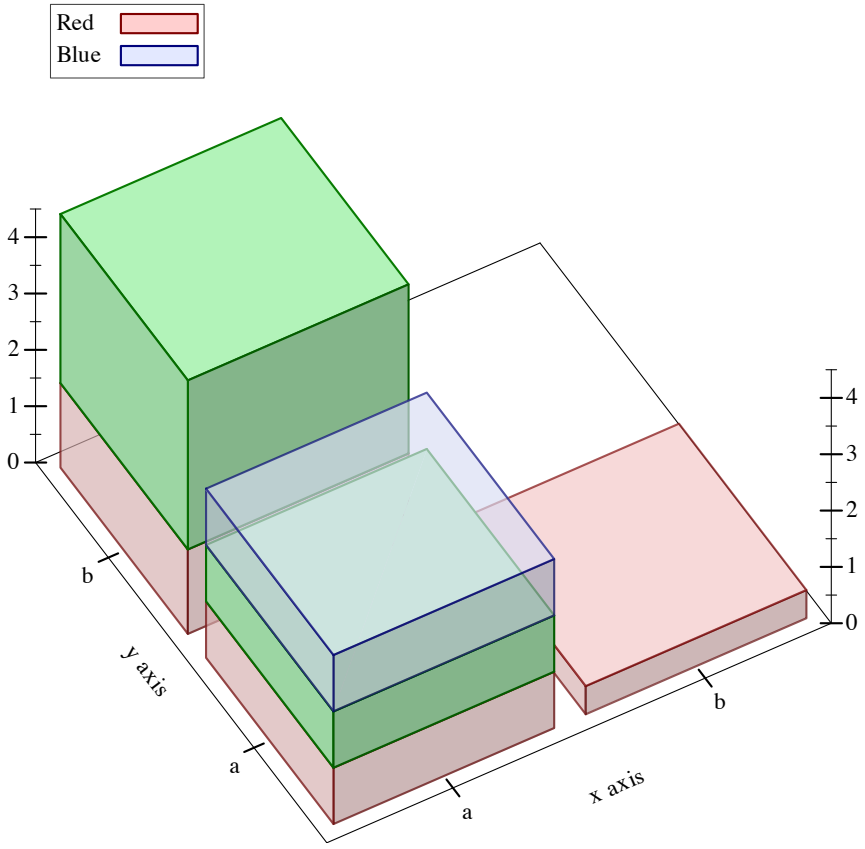

Changed in version 7.9 of package `plot-gui-lib`: Added support for pictures for #:label

```
(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 : (sequence/c (or/c (vector/c any/c any/c (sequence/c real?))
                               (list/c any/c any/c (sequence/c 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)))
```



```
(point-label3d  v
                [label
                 #:color color
                 #:size size
                 #:face face
                 #: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])
 → renderer3d?
  v : (sequence/c real?)
  label : (or/c string? #f) = #f
  color : plot-color/c = (plot-foreground)
  size : (>=/c 0) = (plot-font-size)
  face : (or/c string? #f) = (plot-font-face)
  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. Analogous to point-label.

# 5 Nonrenderers

```
(require plot)
```
package: `plot-gui-lib`

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

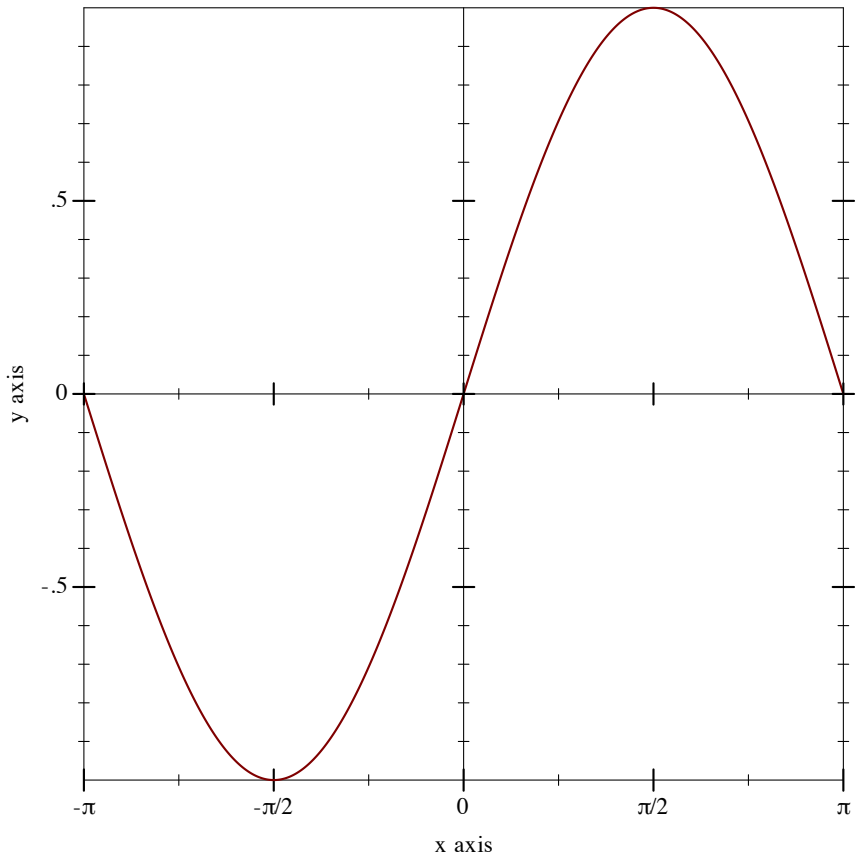
```
(x-ticks ts [#:far? far?]) → nonrenderer?
  ts : (listof tick?)
  far? : boolean? = #f
(y-ticks ts [#:far? far?]) → nonrenderer?
  ts : (listof tick?)
  far? : boolean? = #f
(z-ticks ts [#:far? far?]) → nonrenderer?
  ts : (listof tick?)
  far? : boolean? = #f
```

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))))
```



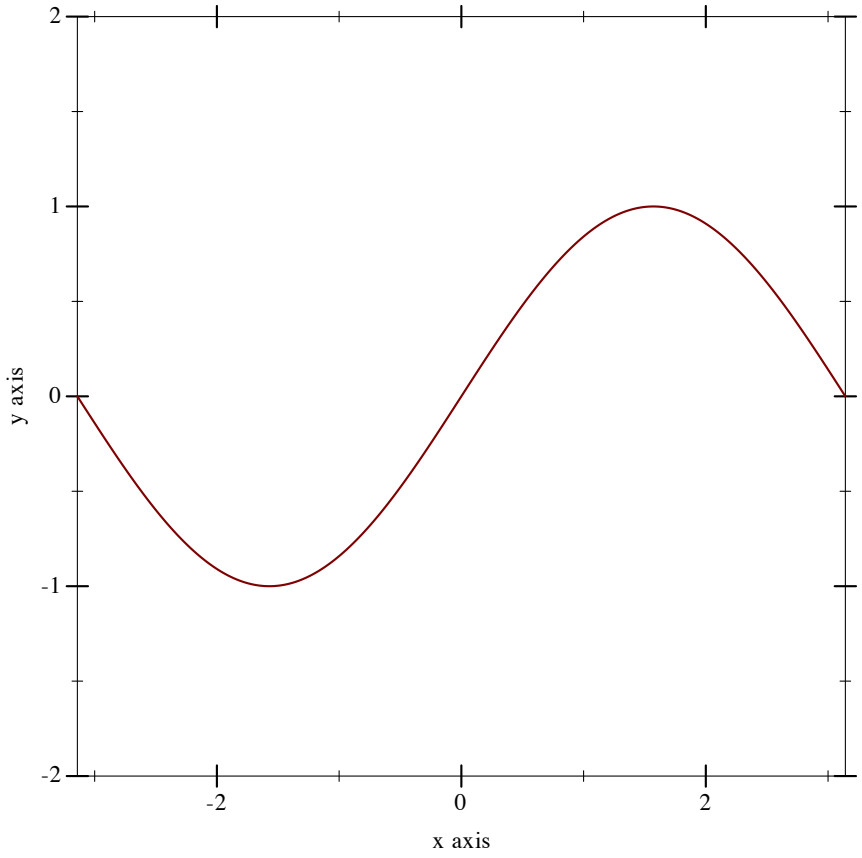

When considering using one of these functions, remember that minor tick labels are never drawn, and that including a `z-ticks` nonrenderer will not add extra contour lines to contour plots.

```
(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.

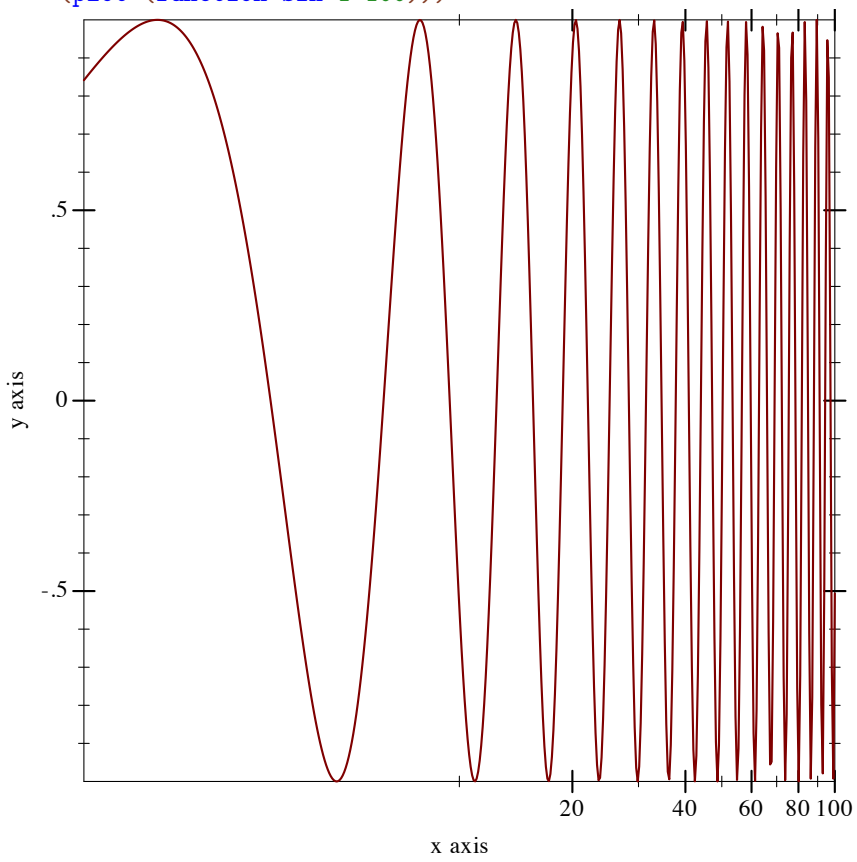
# 6 Axis Transforms and Ticks

```
(require plot)
```
package: `plot-gui-lib`

## 6.1 Axis Transforms

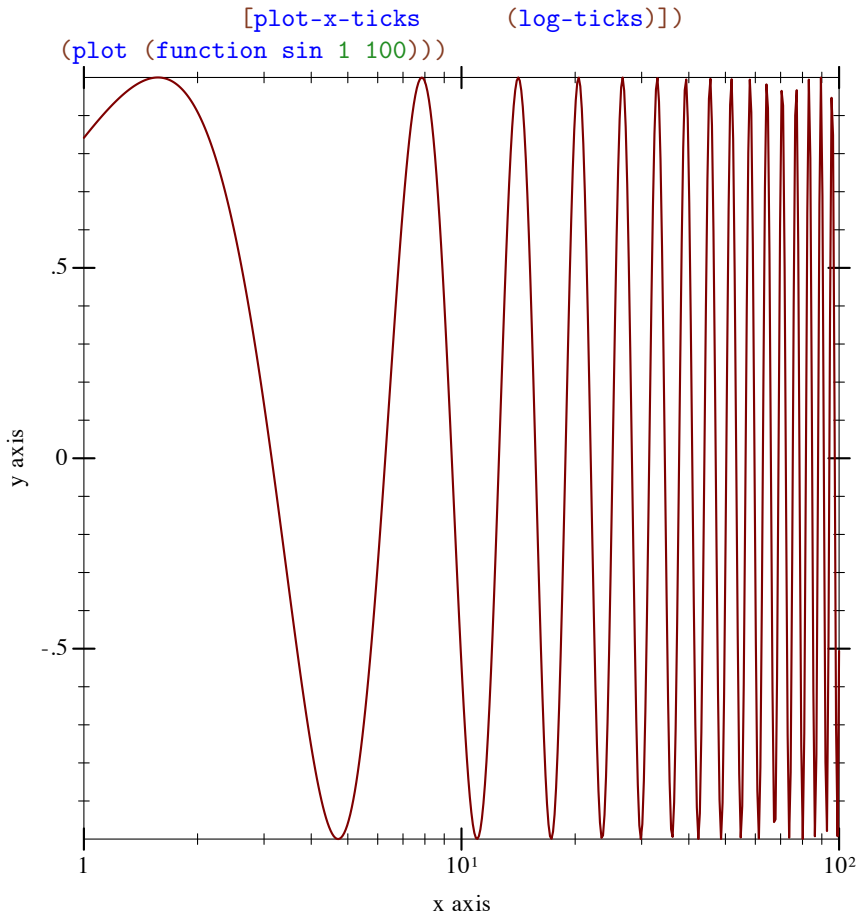
The *x*, *y* and *z* axes for any plot can be independently transformed by parameterizing the plot on different `plot-x-transform`, `plot-y-transform` and `plot-z-transform` values. For example, to plot the *x* axis with a log transform:

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




Most `log-transform`ed 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]
```

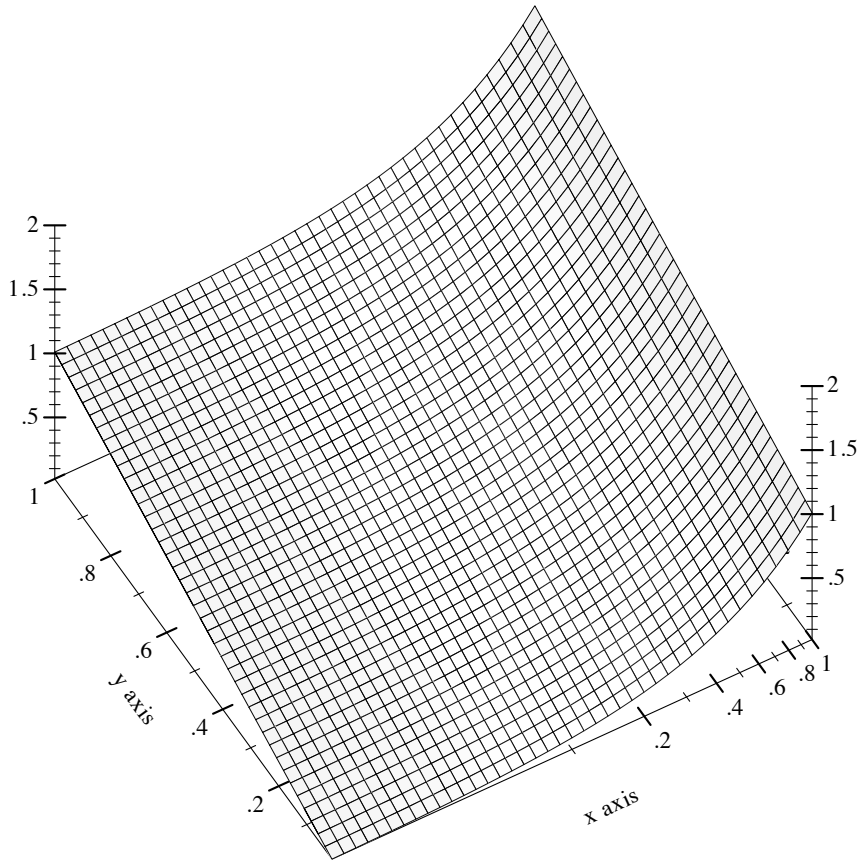



See §6.2 “Axis Ticks” for more details on parameterizing a plot’s axis ticks.

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

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

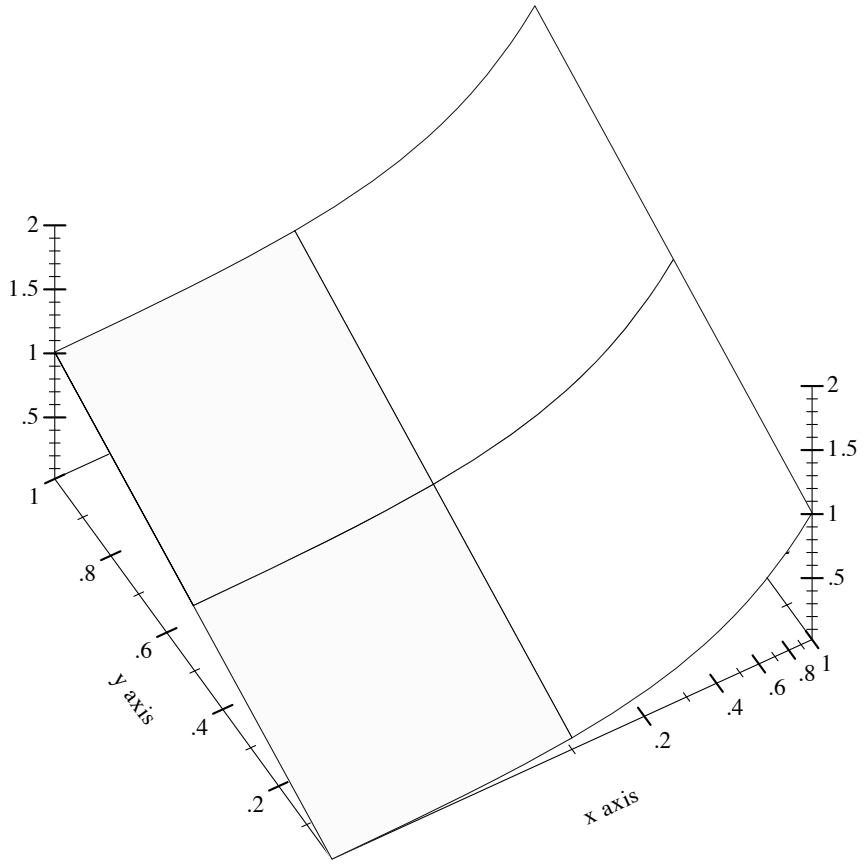
To sample nonlinearly, the *inverse* of a transform is applied to linearly sampled points. See `make-axis-transform` and `nonlinear-seq`.




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 §6 “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
```

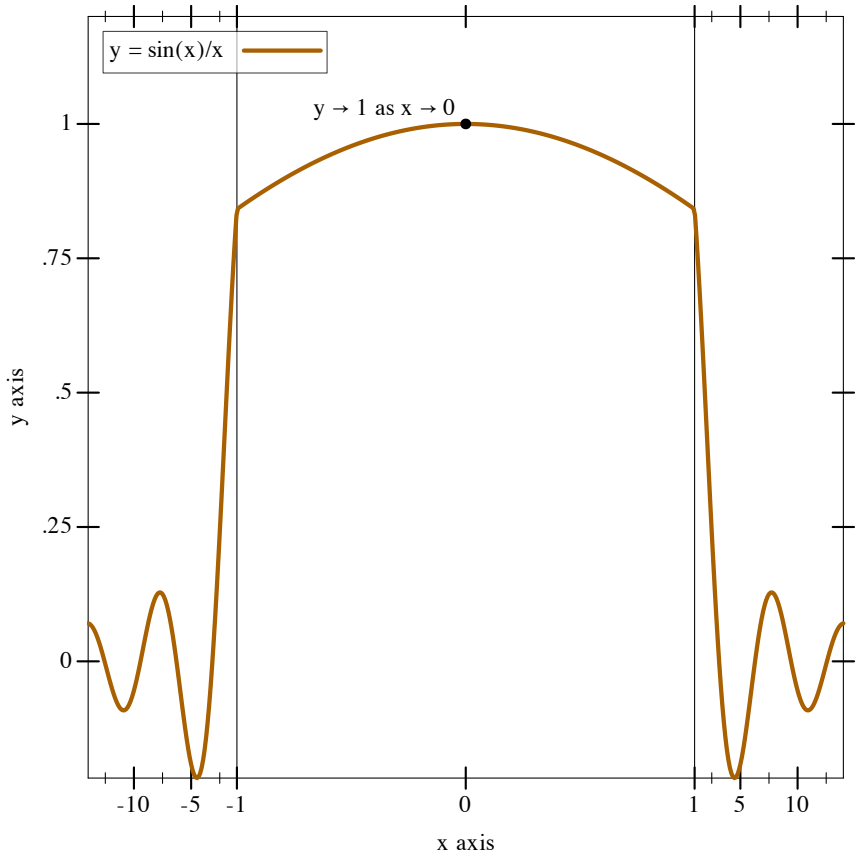
See axis-transform-bound and axis-transform-append for ways to get around an axis transform’s bounds limitations.

```
(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)*):

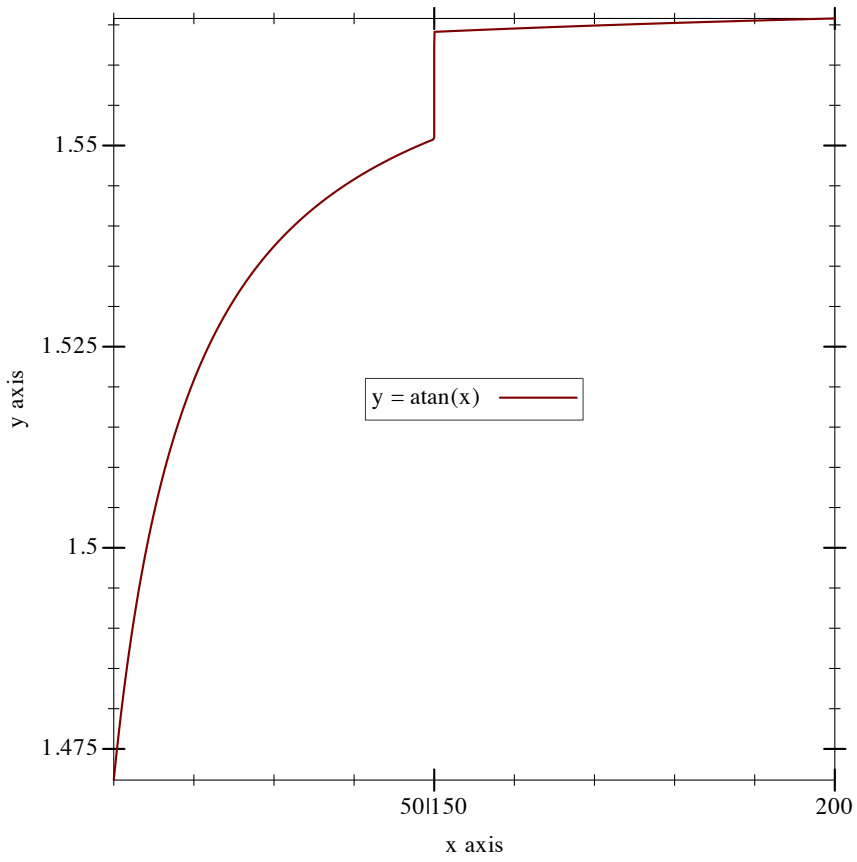
```
> (parameterize ([plot-x-transform  (stretch-transform -1 1 20)]
                 [plot-x-ticks  (ticks-add (plot-x-
ticks) '(-1 1))])
    (plot (list (y-axis -1 #:ticks? #f) (y-axis 1 #:ticks? #f)
                (function (λ (x) (/ (sin x) x)) -14 14
                          #:width 2 #:color 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
```

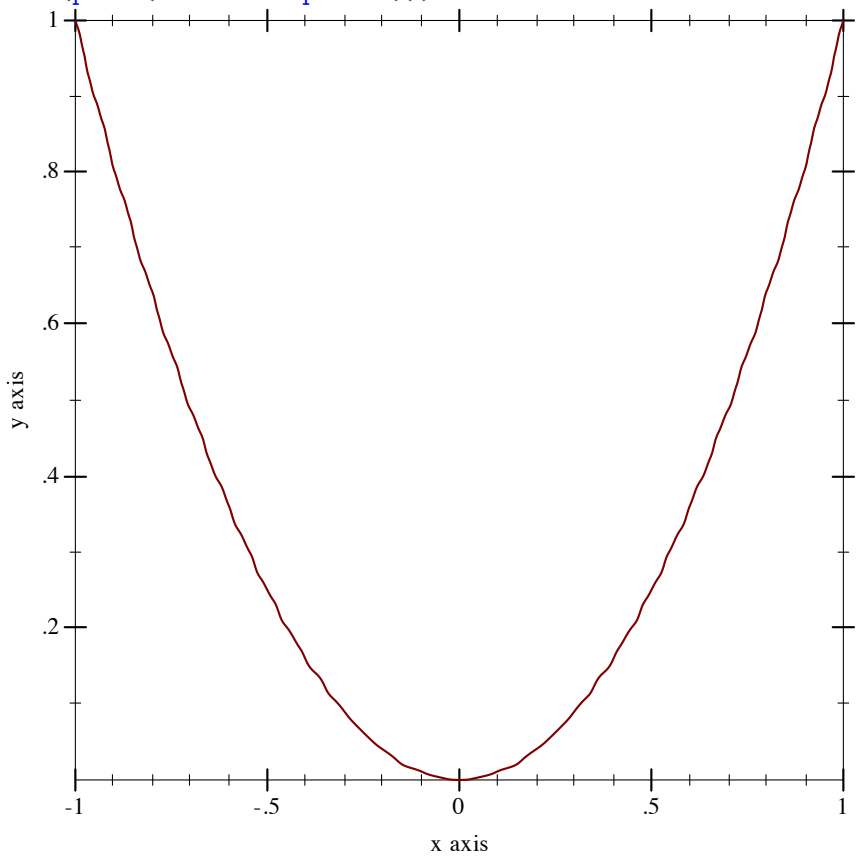
A "cube-root" transform, mostly used for testing. Unlike the log transform, it is defined on the entire real line, making it better for testing the appearance of plots with nonlinearly transformed axes.

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

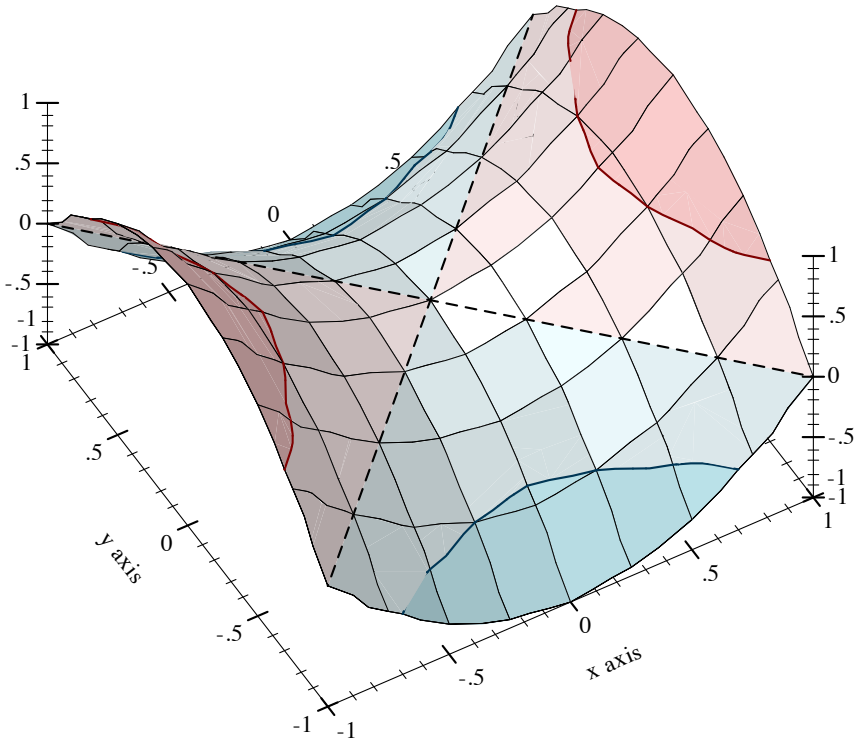
An *extremely important* test case, which makes sure that Plot can use any monotone, invertible function as an axis transform. The *freq* parameter controls the "shakiness" of the transform. At high values, it makes plots look like Peanuts cartoons.

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.

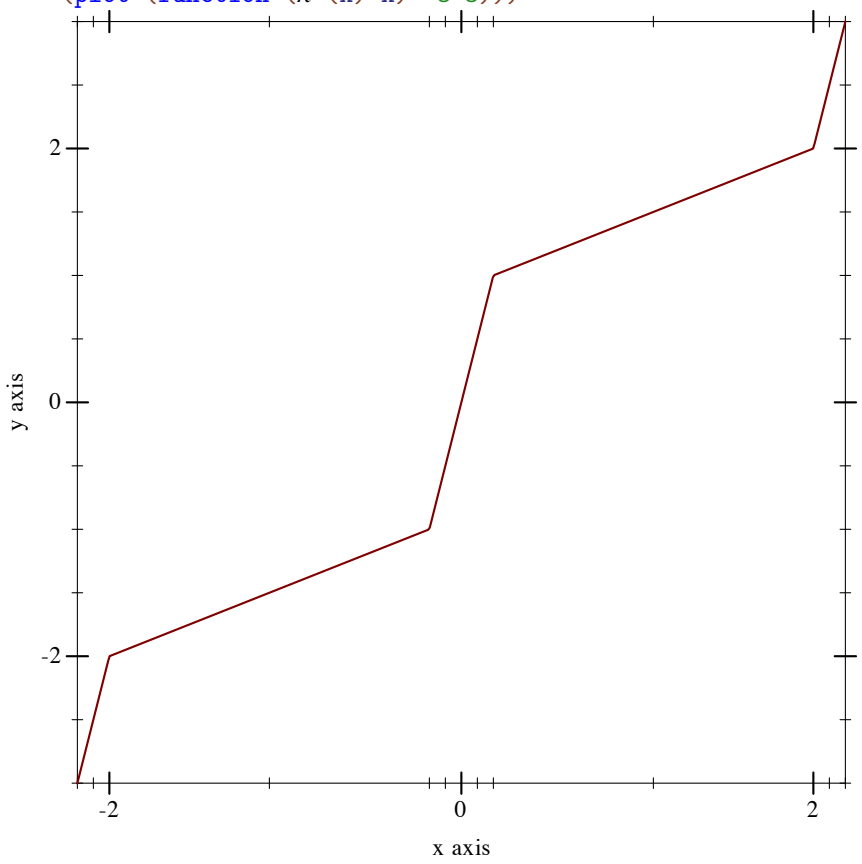
The easiest ways to construct novel axis transforms are to use the axis transform combinators `axis-transform-append`, `axis-transform-bound` and `axis-transform-compose`, or to apply `make-axis-transform` to an `invertible-function`.

```
(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)))
```

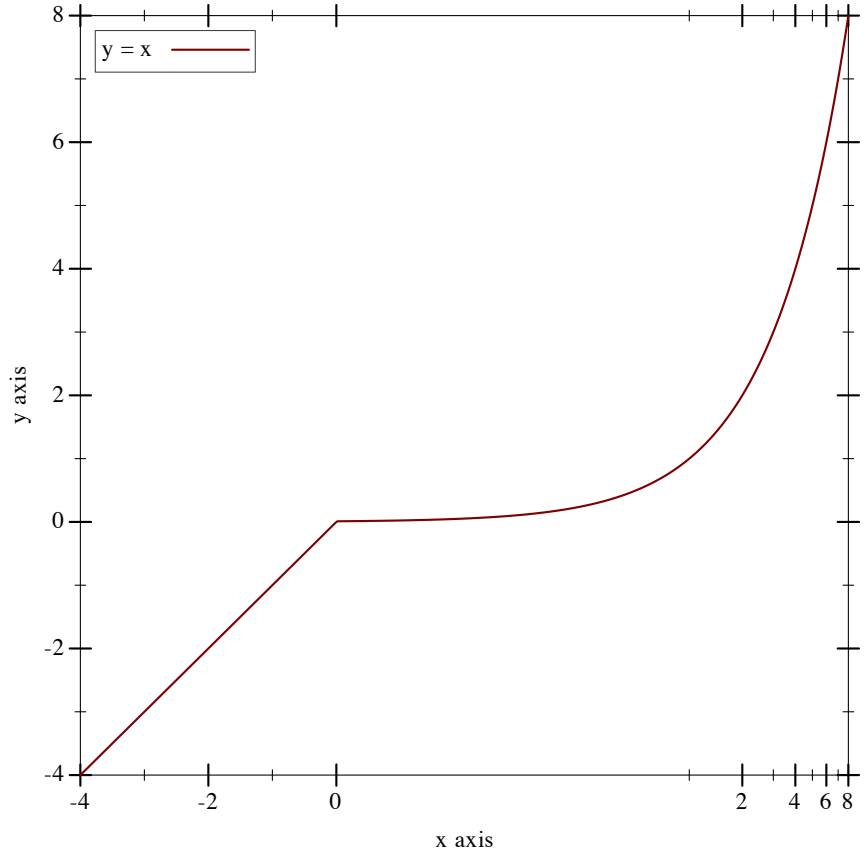



```
(axis-transform-bound t a b) → axis-transform/c
  t : axis-transform/c
  a : real?
  b : real?
```

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
```

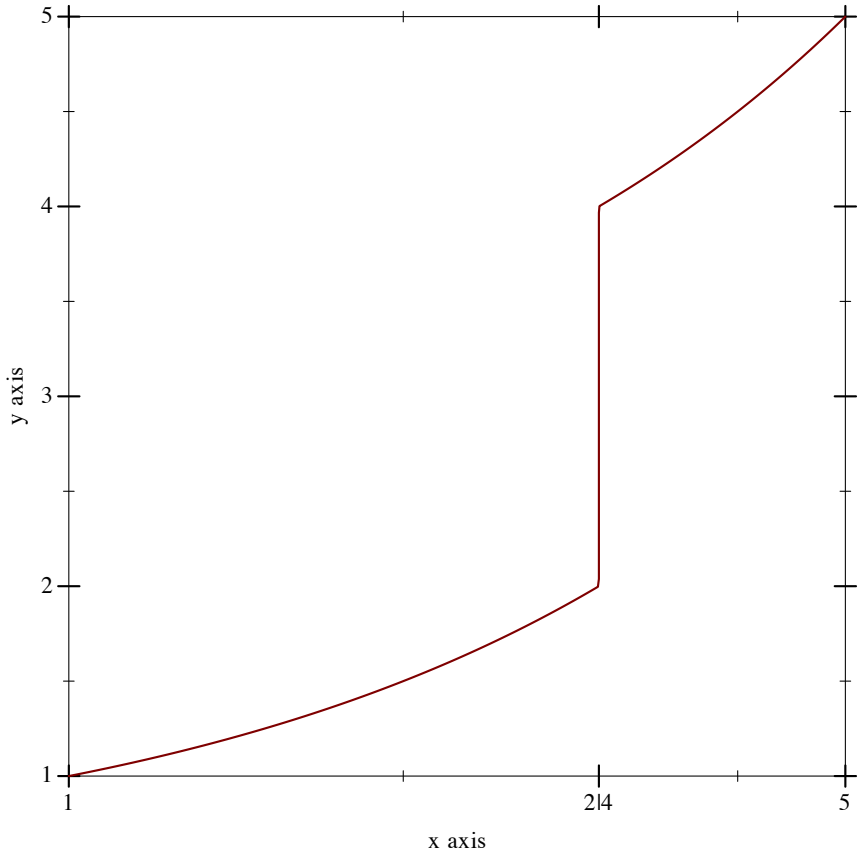
```
                                        log-transform 0.01 +inf.0)])
 (plot (function (λ (x) x) -4 8 #:label "y = x")))
```




```
(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)))
```



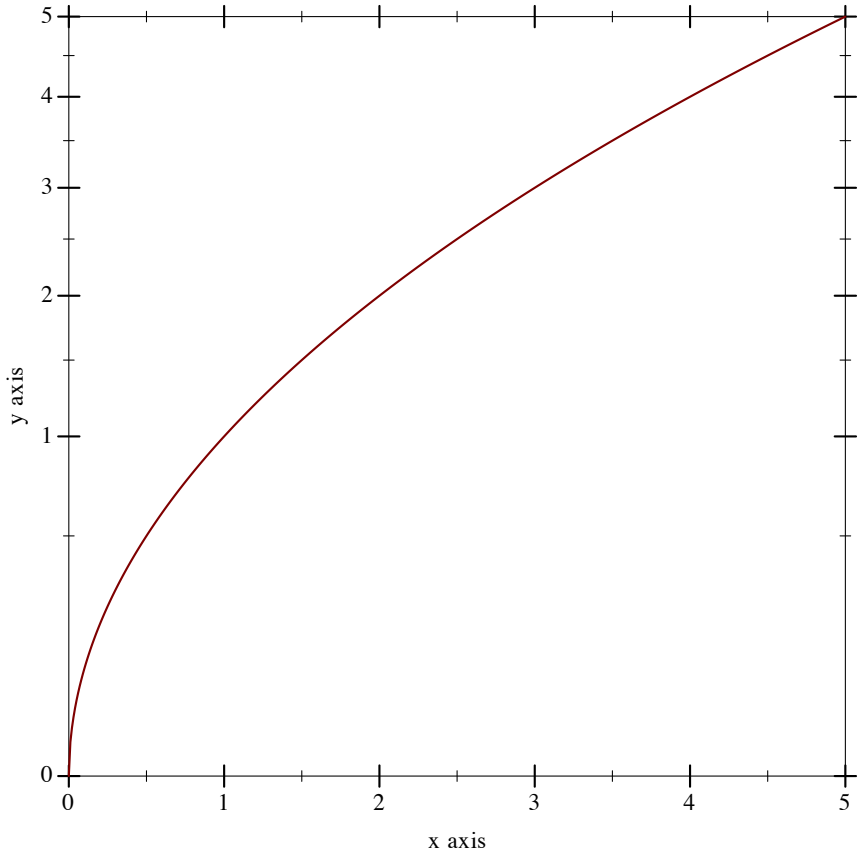

Argument order matters, but predicting the effects of exchanging arguments can be difficult. Fortunately, the effects are usually slight.

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

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

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.

## 6.2 Axis Ticks

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

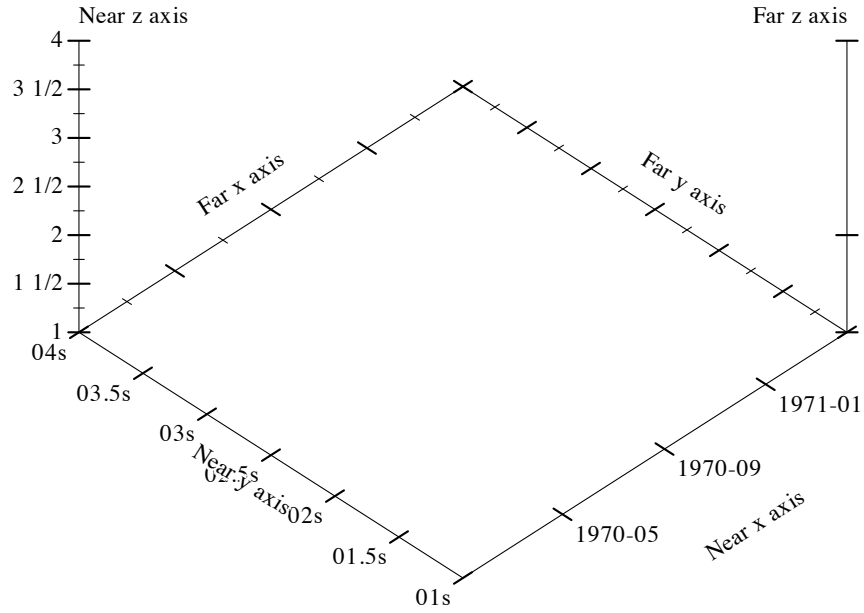
```
(plot-x-ticks) → ticks?
(plot-x-ticks ticks) → void?
  ticks : ticks?
 = (linear-ticks)
(plot-x-far-ticks) → ticks?
(plot-x-far-ticks ticks) → void?
  ticks : ticks?
 = (ticks-mimic plot-x-ticks)
(plot-y-ticks) → ticks?
(plot-y-ticks ticks) → void?
  ticks : ticks?
 = (linear-ticks)
(plot-y-far-ticks) → 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"))
```

Axis Names and Tick Locations



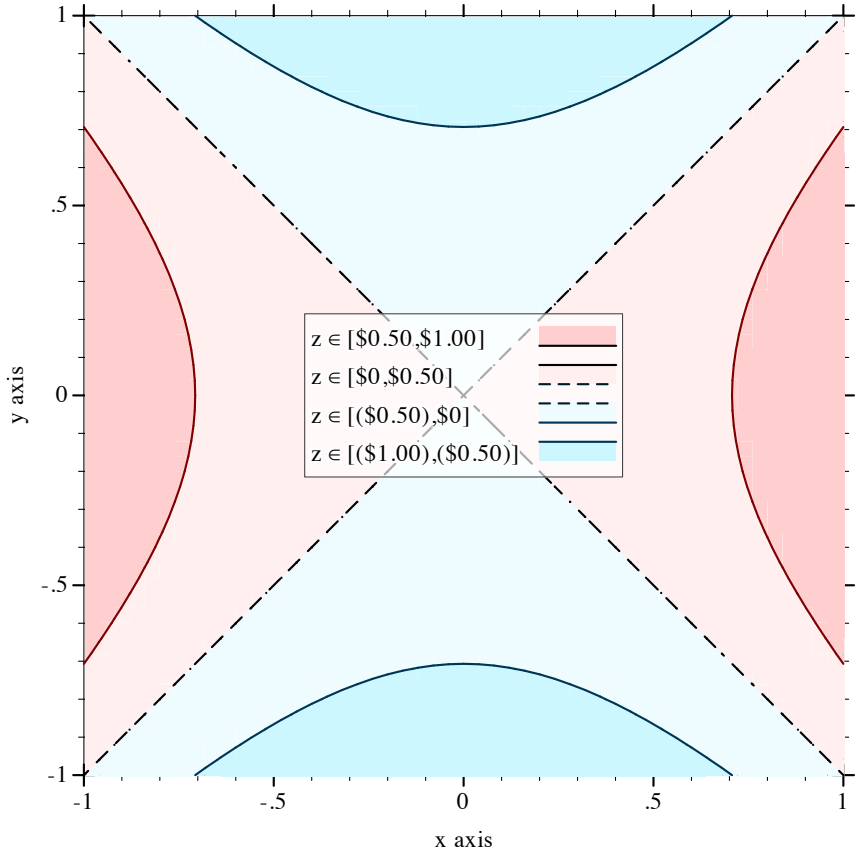

At any `#:angle`, the far *x* and *y* ticks are behind the plot, and the far *z* ticks are on the right. Far ticks are drawn, but not labeled, if they are identical to their corresponding near ticks. They are always identical by default.

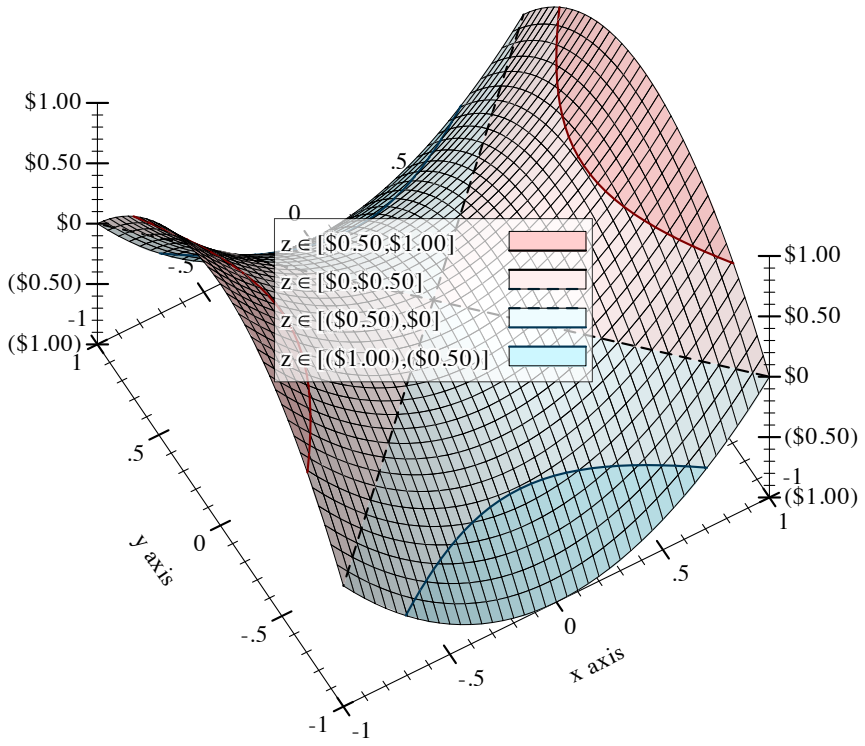
*Major ticks* are longer than *minor ticks*. Major tick labels are always drawn unless collapsed

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 rendered 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 `iso-surfaces3d`, 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? (ticks-generate (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-generate ticks min max) → (listof tick?)
  ticks : ticks?
  min : real?
  max : real?
```

Generates the `tick` values for the range [*`min`*, *`max`*], with layout and format specified by *`ticks`*.

Example:

```
> (ticks-generate (plot-x-ticks) 1/3 2/3)
(list
 (tick 7/20 #f ".35")
 (tick 2/5 #t ".4")
```

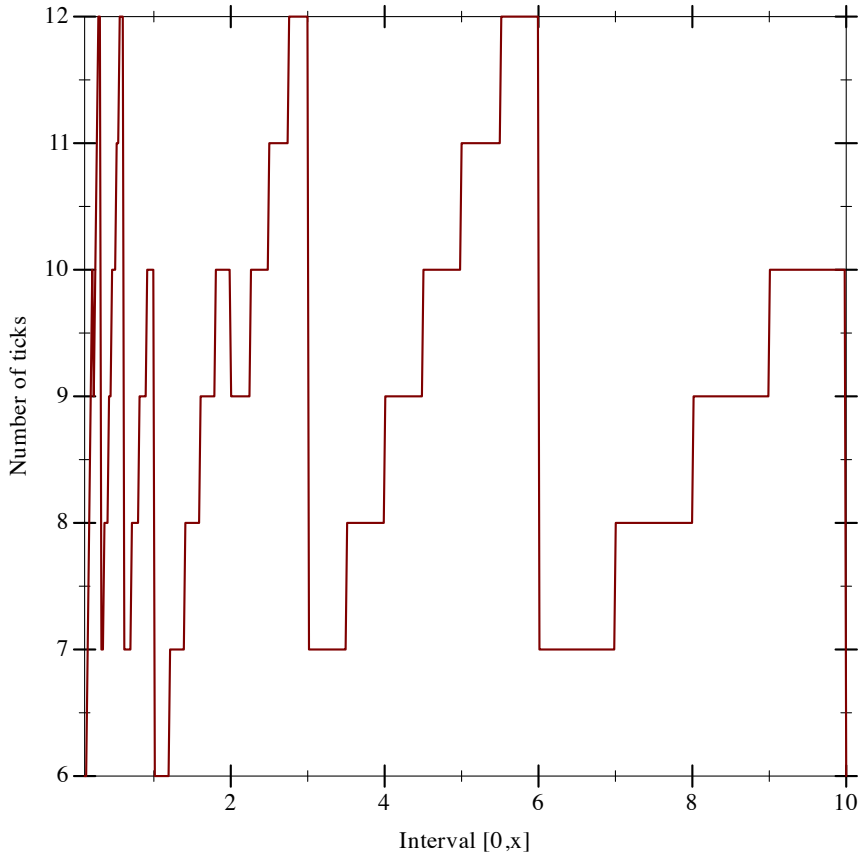
```
 (tick 9/20 #f ".45")
 (tick 1/2 #t ".5")
 (tick 11/20 #f ".55")
 (tick 3/5 #t ".6")
 (tick 13/20 #f ".65"))
```

```
(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")
```




### 6.2.1 Linear Ticks

```
(linear-ticks-layout [#:number number
                      #:base base
                      #:divisors divisors]
                      #:scientific? scientific?)
 → 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)
  scientific? : #t
(linear-ticks-format #:scientific? scientific?)
 → ticks-format/c
  scientific? : #t
```

```
(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)
```

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.

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

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

Changed in version 1.1 of package `plot-gui-lib`: Added the `#:scientific?` argument to `linear-ticks-format` and `linear-ticks`.

### 6.2.2 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]
                  #:scientific? scientific?) → ticks-format/c
  base : (and/c exact-integer? (>=/c 2)) = 10
  scientific? : #t
(log-ticks [#:number number
            #:base base]
            #:scientific? scientific?) → ticks?
  number : exact-positive-integer? = (ticks-default-number)
  base : (and/c exact-integer? (>=/c 2)) = 10
  scientific? : #t
```

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-transform`ed axes, because when exponentially spaced tick positions are `log-transform`ed, they become uniformly spaced.

The `#:base` keyword argument is the logarithm base. The `#:scientific` keyword argument disables scientific formatting, similarly to `linear-ticks`. See `plot-z-far-ticks`

for an example of use.

### 6.2.3 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)
```

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")
```

### 6.2.4 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)
```

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")
```

### 6.2.5 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)
```

The format function and combined `ticks` for uniformly spaced ticks with currency labels; `currency-ticks` uses `linear-ticks-layout` for layout.

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 omitted 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.

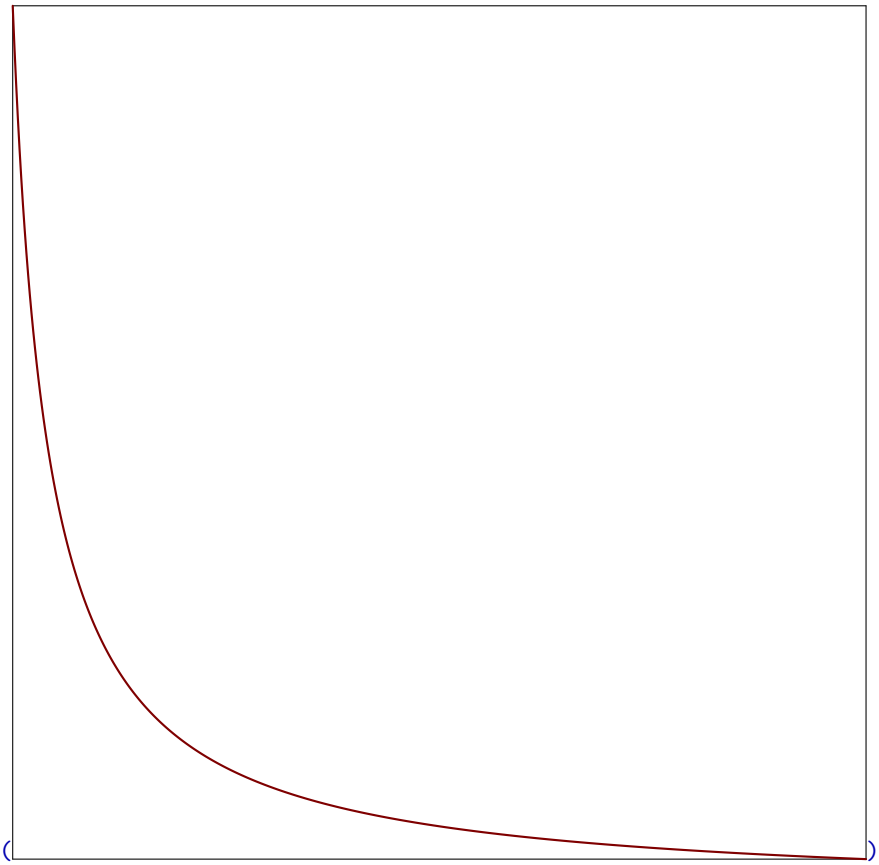
### 6.2.6 Other Ticks

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

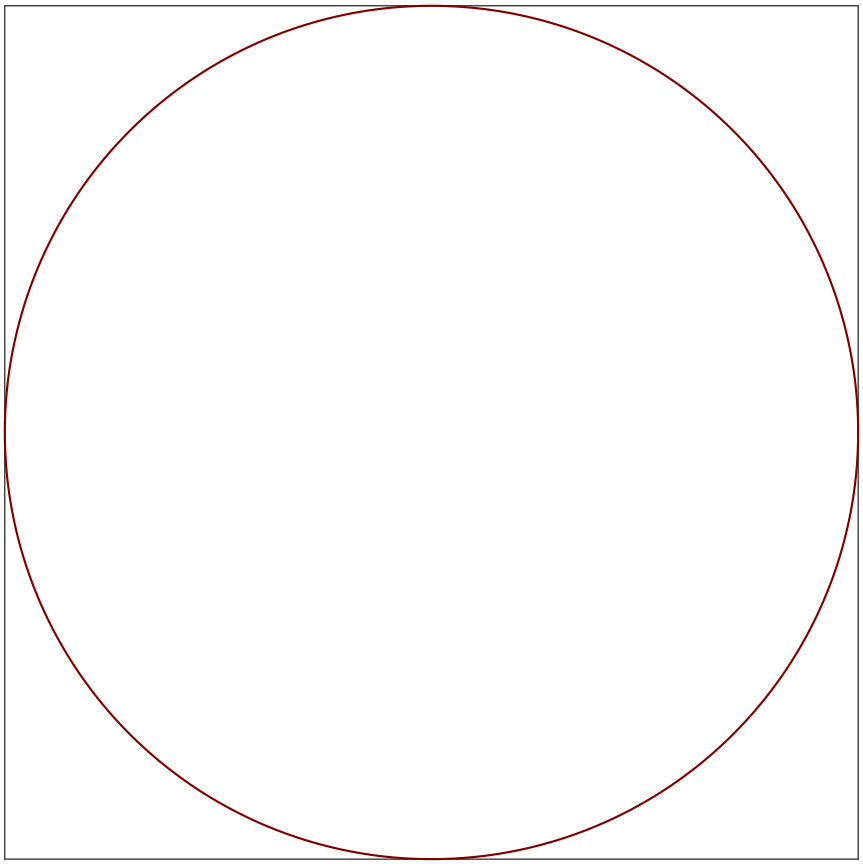
The layout function, format function, and combined `ticks` for no ticks whatsoever.

Examples:

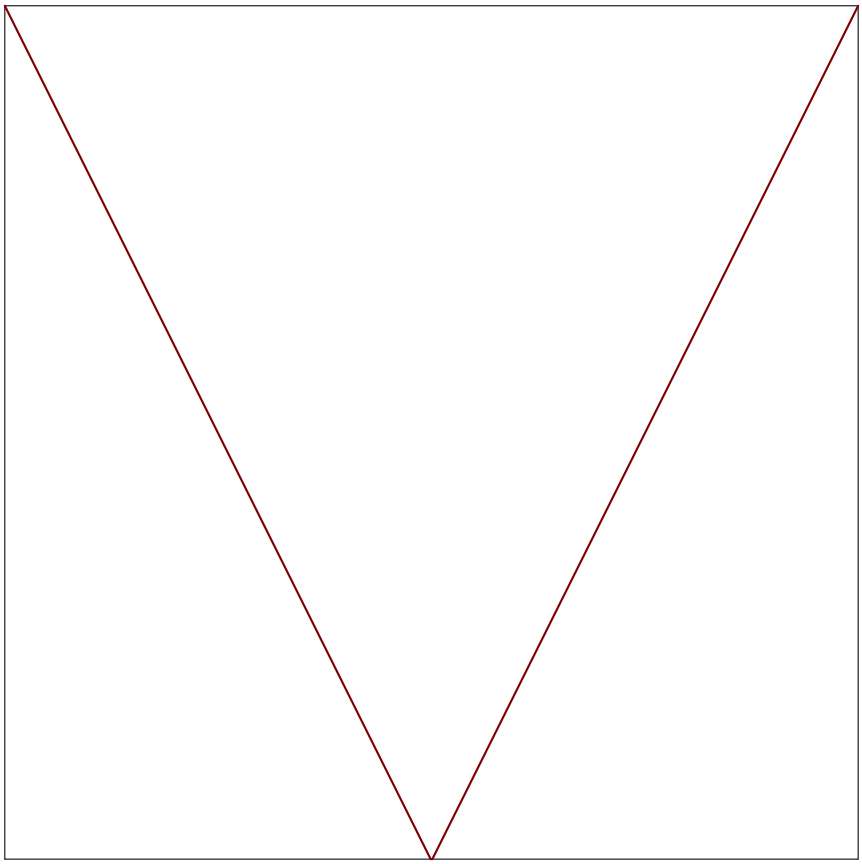
```
> (parameterize ([plot-x-ticks no-ticks]
                 [plot-y-ticks no-ticks]
                 [plot-x-label #f]
                 [plot-y-label #f])
    (list (plot (function /)
                #:x-min 0.01 #:x-max 1/4)))
```



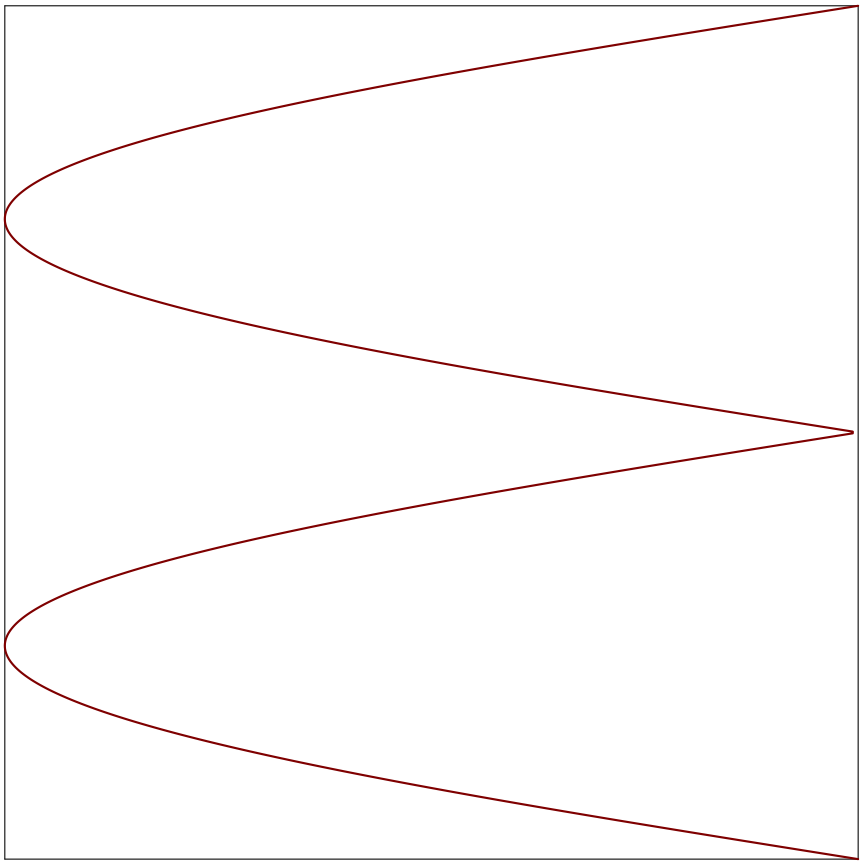
```
'(                                                  )
> (parameterize ([plot-x-ticks  no-ticks]
                 [plot-y-ticks  no-ticks]
                 [plot-x-label  #f]
                 [plot-y-label  #f])
    (plot (polar (λ (θ) 1/3))))
```



```
> (parameterize ([plot-x-ticks no-ticks]
                 [plot-y-ticks no-ticks]
                 [plot-x-label #f]
                 [plot-y-label #f])
    (plot (function (λ (x) (abs (* 2 x))))
          #:x-min -10 #:x-max 10))
```



```
> (parameterize ([plot-x-ticks no-ticks]
                 [plot-y-ticks no-ticks]
                 [plot-x-label #f]
                 [plot-y-label #f])
    (plot (inverse (λ (y) (* -3 (abs (sin y)))))
          #:y-min 0 #:y-max (* 2 pi)))
```



```
(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
```

The format function 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.

For layout, `bit/byte-ticks` uses `linear-ticks-layout` with

- If *kind* is 'SI, base 10 and divisors '(1 2 4 5).
- If *kind* is 'CS, base 2 and divisors '(1 2).

```
(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)
```

The format function and combined `ticks` for fraction-formatted values. For layout, `fraction-ticks` uses `linear-ticks-layout`, passing it the given *divisors*.

### 6.2.7 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
```

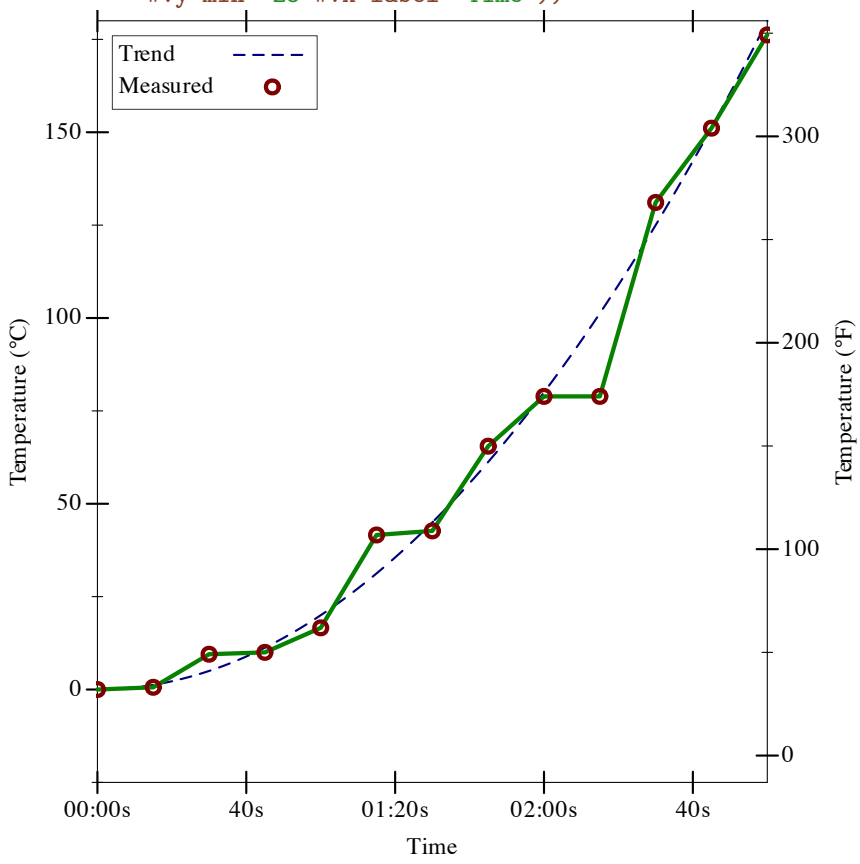
Returns a new `ticks` that acts like `t`, except that it puts additional ticks at positions *xs*. If *major?* is true, the ticks at positions *xs* are all major ticks; otherwise, they are minor ticks.

```
(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 §6.1 “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 Fahrenheit 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"))
```



### 6.2.8 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 in `ticks` structures. The function receives axis bounds and returns a list of `pre-tick`s to be shown on the axis.

Note that the layout function returns `pre-tick`s, or unlabeled ticks, and a separate format function is used to produce the labels for the ticks.

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

The contract for tick format functions in `ticks` structures. The format function receives axis bounds and a list of `pre-tick`s. It must return a label for each `pre-tick` in this list.

The returned labels should be usually distinct, as the plot library will consider ticks with labels that are `string=?` to be duplicates and collapse them, however, this feature can be used by a custom format function to force removal of some ticks from the plot.

Axis bounds can be used to determine how many decimal digits to display, usually by applying `digits-for-range` to the bounds.

## 6.3 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 §6.1 “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?
 = (invertible-function (λ (x) x) (λ (x) x))
```

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?
```

Returns the inverse of an invertible function.

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

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

# 7 Plot Utilities

```
(require plot/utils)
```
package: plot-lib

## 7.1 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?]) → string?
  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).

## 7.2 Sampling

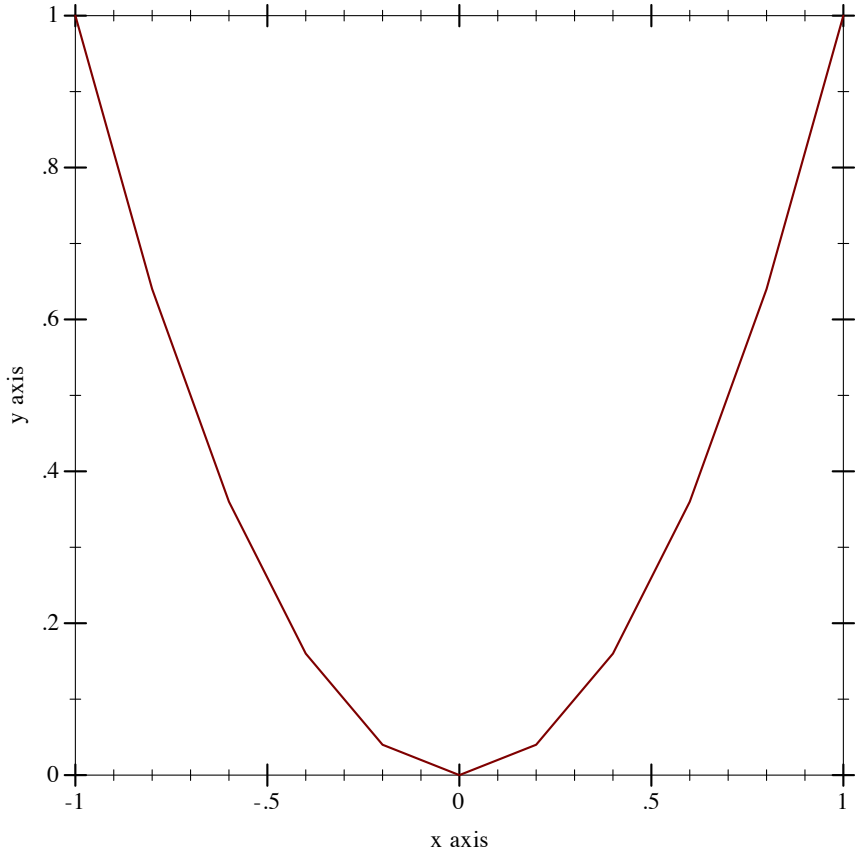
```
(linear-seq  start
             end
             num
            [#:start? start?
             #:end? end?])    → (listof real?)
  start : real?
  end : real?
  num : exact-nonnegative-integer?
  start? : boolean? = #t
  end? : boolean? = #t
```

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:

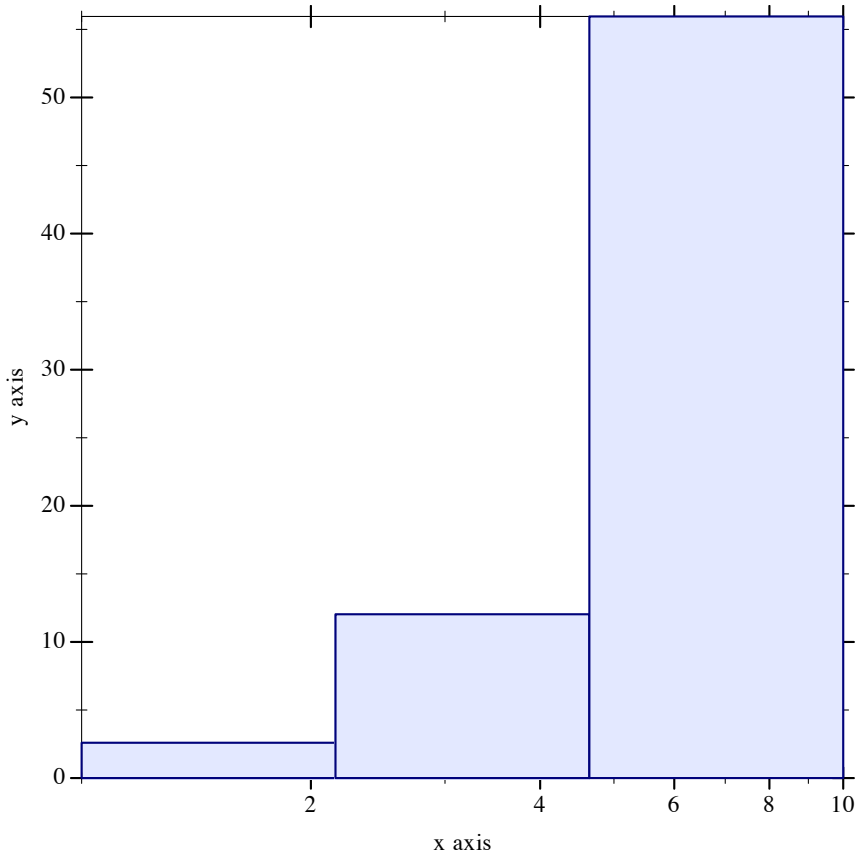
```
> (linear-seq* '(0 1 2) 5)
'(0 1/2 1 3/2 2)
> (linear-seq* '(0 1 2) 6)
'(0 2/5 4/5 6/5 8/5 2)
> (linear-seq* '(0 1 0) 5)
'(0 1/2 1 1/2 0)
```

```
(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))))
```




```
(kde xs h [ws]) → (-> real? real?)
                  (or/c rational? #f)
                  (or/c rational? #f)
  xs : (listof real?)
  h : (>/c 0)
  ws : (or/c (listof (>=/c 0)) #f) = #f
```

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

```
(silverman-bandwidth xs) → real?
  xs : (listof real?)
```

Returns the Silverman Bandwidth estimator for the given samples. This is used as the default bandwidth by the `violin` renderer. See the kernel density estimation Wikipedia article for more details.

Added in version 8.5 of package plot-lib.

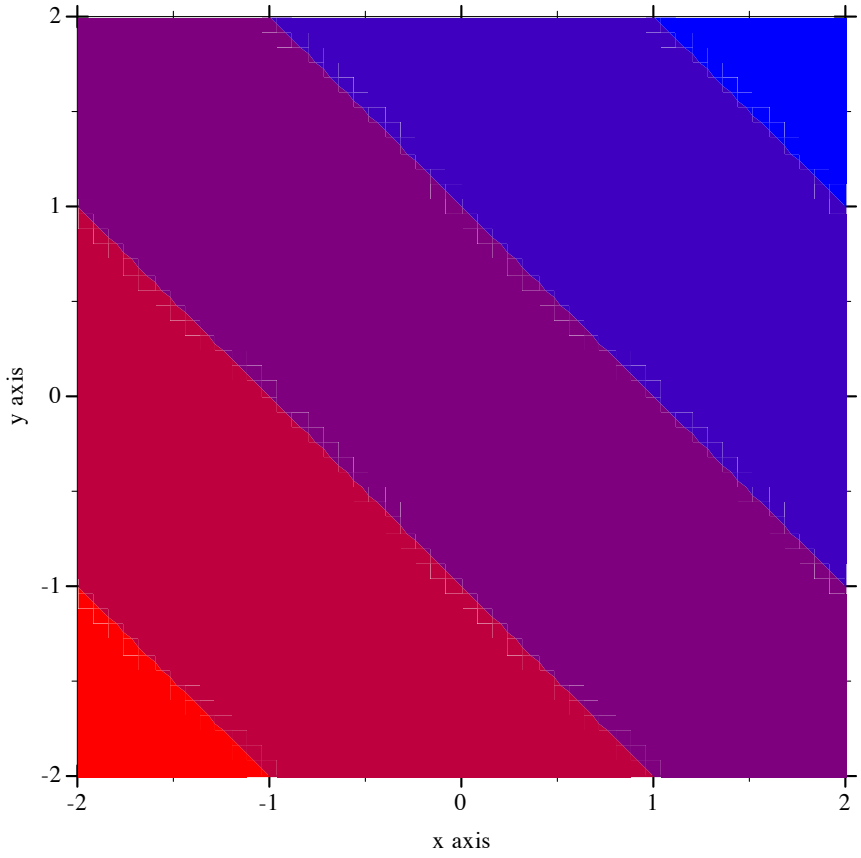
## 7.3 Plot Colors and Styles

```
(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)))
```



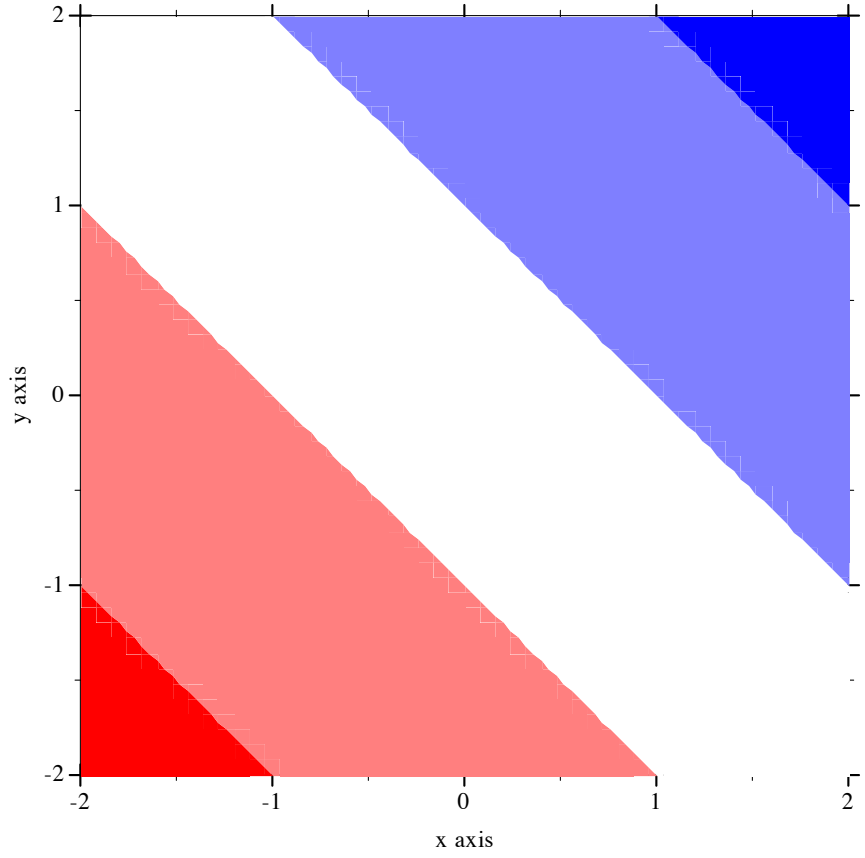

```
(color-seq* colors
            num
            [#:start? start?
             #:end? end?])
 → (listof (list/c real? real? real?))
  colors : (listof color/c)
  num : exact-nonnegative-integer?
  start? : boolean? = #t
  end? : boolean? = #t
```

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)
```

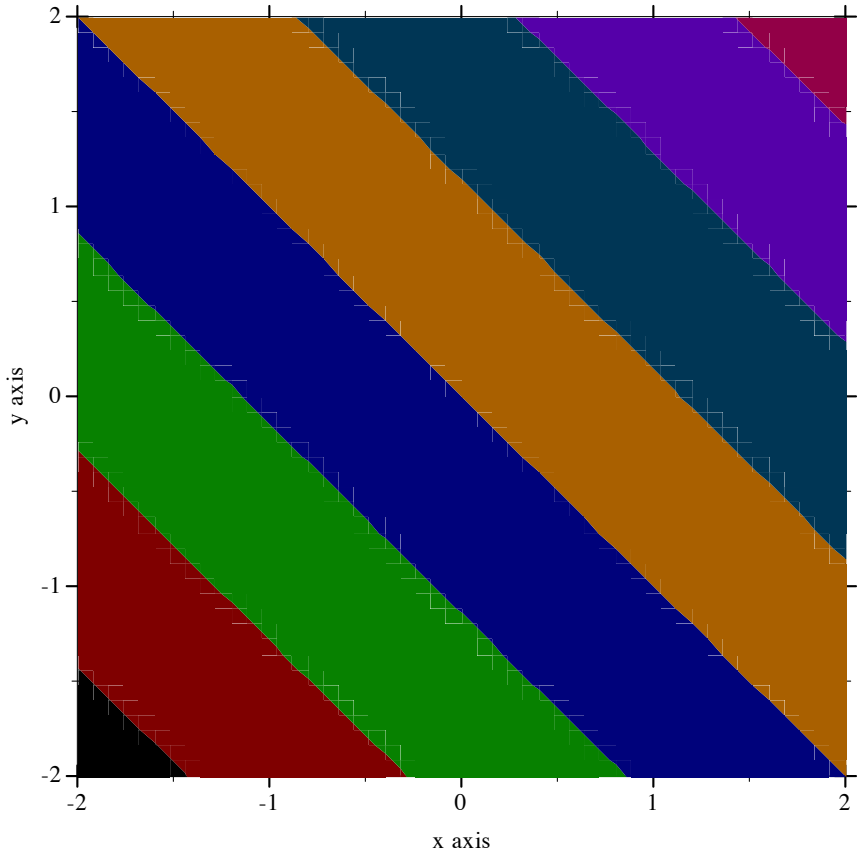
This function does not convert integers to RGB triplets, because there is no way for it to know whether the color will be used for a pen or for a brush. Use `->pen-color` and `->brush-color` to convert integers.

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

Convert a *line* color to an RGB triplet. Integer colors are looked up in the current `plot-pen-color-map`, and non-integer colors are converted using `->color`. When the integer color is larger than the number of colors in the color map, it will wrap around.

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))))
```



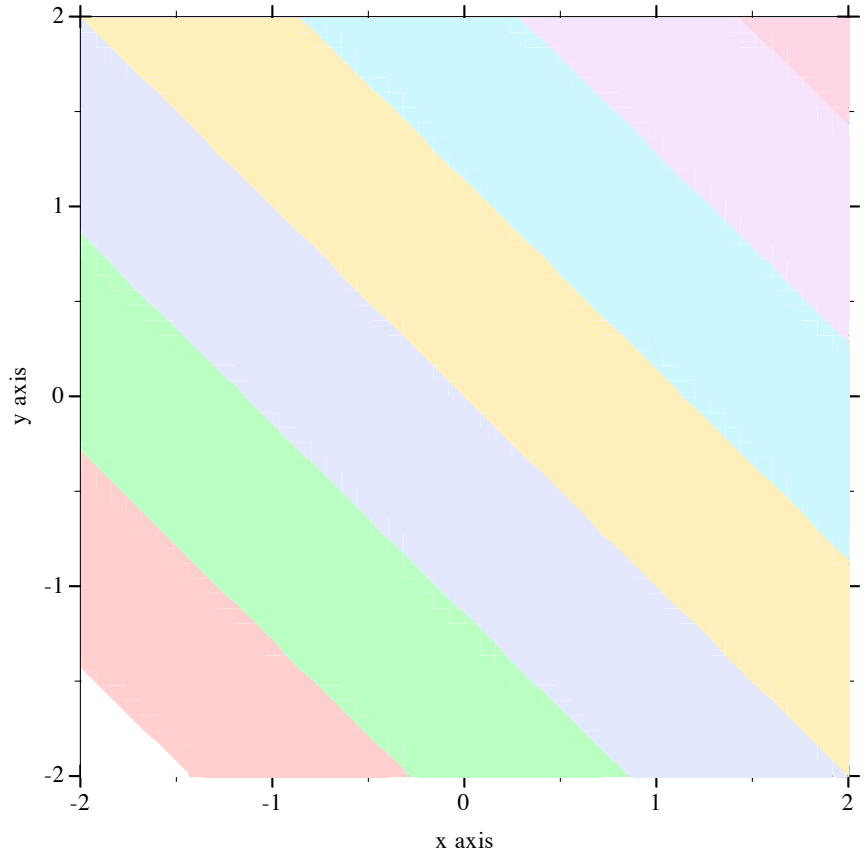

The example above is using the internal color map, with `plot-pen-color-map` set to `#f`.

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

Convert a *fill* color to an RGB triplet. Integer colors are looked up in the current `plot-brush-color-map` and non-integer colors are converted using `->color`. When the integer color is larger than the number of colors in the color map, it will wrap around.

Examples:

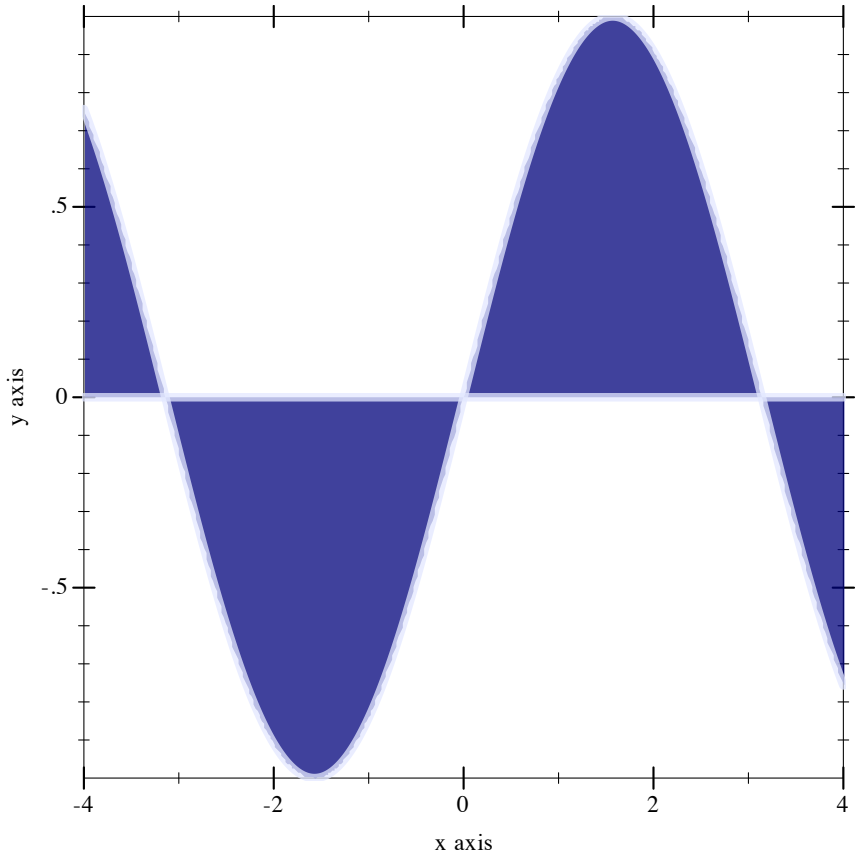
```
> (equal? (->brush-color 0) (->brush-color 8))
#f
> (plot (contour-intervals
         (λ (x y) (+ x y)) -2 2 -2 2
         #:levels 7 #:contour-styles '(transparent)
         #:colors (map ->brush-color (build-list 8 values))))
```




The example above is using the internal color map, with `plot-brush-color-map` is set to `#f`. In this 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)
```

```
(color-map-names) → (listof symbol?)
```

Return the list of available color map names to be used by `plot-pen-color-map` and `plot-brush-color-map`.

Added in version 7.3 of package `plot-lib`.

```
(color-map-size name) → integer?
  name : symbol?
```

Return the number of colors in the color map *name*. If *name* is not a valid color map name, the function will signal an error.

Added in version 7.3 of package `plot-lib`.

```
(register-color-map name color-map) → void
  name : symbol?
  color-map : (vectorof (list byte? byte? byte?))
```

Register a new color map *name* with the colors being a vector of RGB triplets. If a color map by that name already exists, it is replaced.

Added in version 7.3 of package `plot-lib`.

## 7.4 Plot-Specific Math

### 7.4.1 Real Functions

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

Converts 2D polar coordinates to 2D 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.

```
(min* r ...) → real?
  r : real?
(max* r ...) → real?
  r : real?
```

Compute the min or max of a sequence of numbers.

Examples:

```
> (min* 3 5 4 2 1)
1
> (max* 3 5 4 2 1)
5
> (min*)
+inf.0
> (max*)
-inf.0
```

### 7.4.2 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))
vrational?: contract violation
  expected: real?
  given: #f
  in: an element of
      the 1st argument of
      (-> (vectorof real?) any)
  contract from:
      <pkgs>/plot-lib/plot/private/common/math.rkt
  blaming: top-level
   (assuming the contract is correct)
  at: <pkgs>/plot-lib/plot/private/common/math.rkt:304:9
```

### 7.4.3 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?
```

## 7.5 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)
```

## 7.6 Plot Metrics

```
plot-metrics<%> : interface?
```

The `plot-metrics<%>` interface allows obtaining plot area positions on the plots returned by `plot`, `plot-snip`, `plot-bitmap`, `plot/dc`, as well as their 3D variants, `plot3d`,

plot3d-snip, plot3d-bitmap and plot3d/dc. All plot objects returned by these functions implement this interface.

For plots created by plot-pict and plot3d-pict, there is a separate set of functions that provide the same functionality, for example, see plot-pict-bounds.

```
(send a-plot-metrics get-plot-bounds)
 → (vectorof (vector/c real? real?))
```

Return the bounds of the plot as a vector of minimum and maximum values, one for each axis in the plot. For 2D plots, this method returns a vector of two elements, for the X and Y axes, while 3D plots return a vector of three elements for the X, Y and Z axes.

The values returned are in plot coordinates, to obtain the coordinates on the drawing surface (i.e. image coordinates), use plot->dc on these bounds.

Plot bounds for interactive plots, like those produced by plot and plot-snip, can change as the user zoom in and out the plot, get-plot-bounds always returns the current bounds of the plot, but they might be invalidated by a user operation.

```
(send a-plot-metrics plot->dc coordinates) → (vectorof real?)
  coordinates : (vectorof real?)
```

Convert *coordinates* from plot coordinate system to the drawing coordinate system (that is, image coordinates). For 2D plots, *coordinates* is a vector of two values, the X and Y coordinates on the plot, while for 3D plots it is a vector of three values, the X, Y and Z coordinates.

This method can be used, for example, to determine the actual location on the image where the coordinates 0, 0 are. It can also be used to determine the location of the plot area inside the image, by calling it on the plot bounds returned by get-plot-bounds.

For interactive plots, the coordinates might change as the user zooms in and out the plot.

```
(send a-plot-metrics dc->plot coordinates) → (vectorof real?)
  coordinates : (vectorof real?)
```

For 2D plots, this method returns the 2D plot coordinates that correspond to the input *coordinates*, which are in the draw context coordinate system.

For 3D plots, this method returns a 3D position on the plane perpendicular to the user view for the plot. Together with the normal vector for this plane, returned by plane-vector, the projection line can be reconstructed.

This is the reverse operation from plot->dc and same remark about the user zooming in and out the plot applies.

```
(send a-plot-metrics plane-vector) → (vectorof real?)
```

Return the unit vector representing the normal of the screen through the plot origin. For 2D plots this always returns `#(0 0 1)`, for 3D plots this unit vector can be used to reconstruct plot coordinates from draw context coordinates.

For interactive 3D plots, the returned value will change if the user rotates the plot.

Added in version 8.1 of package `plot-lib`.

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

Return `#t` if *any* is a plot returned by `plot-pict`. This can be used to determine if the functions `plot-pict-bounds`, `plot-pict-plot->dc`, `plot-pict-dc->plot` and `plot-pict-plane-vector` can be called on it.

Added in version 8.1 of package `plot-lib`.

```
(plot-pict-bounds plot) → (vectorof (vector/c real? real?))
  plot : plot-pict?
```

Return the bounds of the plot returned by `plot-pict`. See `get-plot-bounds` for more details.

Added in version 8.1 of package `plot-lib`.

```
(plot-pict-plot->dc plot coordinates) → (vectorof real?)
  plot : plot-pict?
  coordinates : (vectorof real?)
```

Convert the plot *coordinates* to draw context coordinates for the *plot*. See `plot->dc` for more details.

Added in version 8.1 of package `plot-lib`.

```
(plot-pict-dc->plot plot coordinates) → (vectorof real?)
  plot : plot-pict?
  coordinates : (vectorof real?)
```

Convert the draw contect *coordinates* to plot coordinates for the *plot*. See `dc->plot` for more details.

Added in version 8.1 of package `plot-lib`.

```
(plot-pict-plane-vector plot) → (vectorof real?)
  plot : plot-pict?
```

Return the unit vector representing the normal of the screen through the plot origin. For 2D plots this always returns `#(0 0 1)`, for 3D plots this can be used to reconstruct plot coordinates from draw context coordinates. See `plane-vector` for more details.

Added in version 8.1 of package `plot-lib`.

# 8 Plot and Renderer Parameters

```
(require plot)
```
package: plot-gui-lib

## 8.1 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.

## 8.2 Output

```
(plot-new-window?) → boolean?
(plot-new-window? new-window?) → void?
  new-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). Used for default arguments of plotting procedures such as plot and plot3d.

```
(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%`.

## 8.3 General Appearance

```
(plot-aspect-ratio) → (or/c (and/c rational? positive?) #f)
(plot-aspect-ratio ratio) → void?
  ratio : (or/c (and/c rational? positive?) #f)
 = #f
```

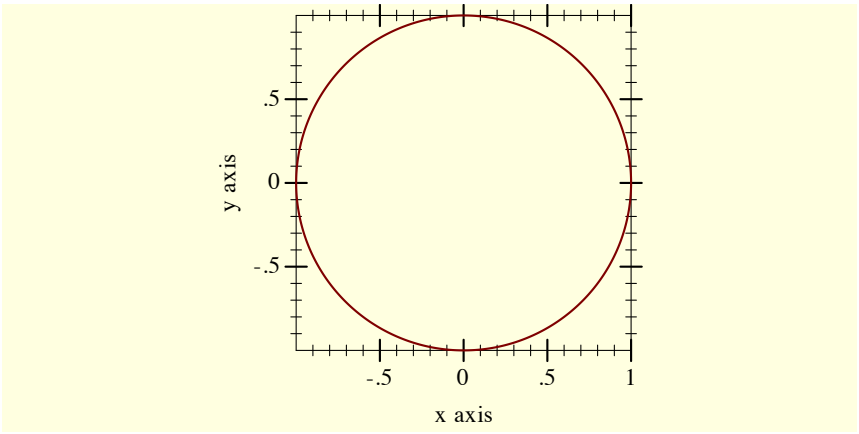
Controls the aspect ratio of the plot area, independently from the width and height of the entire plot.

When the aspect ratio is `#f`, the plot area fill fill the entire area of the plot, leaving room only for the axis labels and title.

When an aspect ratio is a positive number, the plot area will maintain this aspect ratio, possibly leaving empty areas around the plot.

This feature is useful when the aspect ratio needs to be maintained for the plot output to look correct, for example when plotting a circle:

```
> (parameterize ([plot-aspect-ratio 1]
                 [plot-background "lightyellow"])
    (plot (polar (lambda (t) 1)) #:width 400 #:height 200))
```




Added in version 8.1 of package `plot-gui-lib`.

```
(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
```

Title and near 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. Used as default keyword arguments of plotting procedures such as `plot` and `plot3d`.

```
(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
```

Far 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. See `plot-x-ticks` for a discussion of near and far axes.

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

Number of samples taken of functions plotted by 3D renderers, per-axis. Used as the default `#:samples` argument of `surface3d`, `polar3d`, `isoline3d`, `contours3d`, `contour-intervals3d`, `isosurface3d` and `isosurfaces3d`.

```
(plot3d-angle) → real?
(plot3d-angle angle) → void?
  angle : real?
 = 30
(plot3d-altitude) → real?
(plot3d-altitude altitude) → void?
  altitude : real?
 = 60
```

The angle and altitude of the camera in rendering 3D plots, in degrees. Used as default keyword arguments of plotting procedures such as `plot3d`.

```
(plot3d-ambient-light) → (real-in 0 1)
(plot3d-ambient-light amt) → void?
  amt : (real-in 0 1)
 = 2/3
(plot3d-diffuse-light?) → boolean?
(plot3d-diffuse-light? diffuse?) → void?
  diffuse? : boolean?
 = #t
(plot3d-specular-light?) → boolean?
(plot3d-specular-light? specular?) → void?
  specular? : boolean?
 = #t
```

Amount of ambient light, and whether 3D plots are rendered with diffuse and specular reflectance.

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

The width of the lines used to draw plot axes and other non-renderer elements.

The line width for plot renderers, such as `function` and `lines`, is controlled by the `line-width` parameter.

```
(plot-line-cap) → plot-pen-cap/c
(plot-line-cap cap) → void?
  cap : plot-pen-cap/c
 = 'round
```

The cap of the lines used to draw plot axes and other non-renderer elements. See also `line-cap`.

Added in version 8.10 of package `plot-gui-lib`.

```
(plot-inset)
 → (or/c (>=/c 0) (list (>=/c 0) (>=/c 0) (>=/c 0) (>=/c 0)))
(plot-inset inset) → void?
  inset : (or/c (>=/c 0) (list (>=/c 0) (>=/c 0) (>=/c 0) (>=/c 0)))
 = 0
```

The amount of space around the plot to leave unused, when calculating plot layouts for ticks and axis labels. The parameter can be specified as a single value, which applies to all sides of the plot image, or as a list of four separate values for the left, right, top, and bottom margins of the plot image.

One example use for this parameter is to avoid clipping tick marks when lines for plot elements are very thick, see `plot-line-width` and `line-width`. In such a case, the end of axis ticks can be drawn beyond the end point of the line, and might be clipped at the edge of the drawing region. A non-zero `plot-inset` value can be used to avoid this clipping.

See also `plot-legend-padding` for an equivalent setting for the plot legend.

Added in version 8.11 of package `plot-gui-lib`.

```
(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
(plot-font-face) → (or/c string? #f)
(plot-font-face face) → void?
  face : (or/c string? #f)
 = #f
(plot-font-family) → font-family/c
(plot-font-family family) → void?
  family : font-family/c
 = 'roman
```

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

```
(plot-legend-font-size) → (or/c (>=/c 0) #f)
(plot-legend-font-size size) → void?
  size : (or/c (>=/c 0) #f)
 = #f
(plot-legend-font-face) → (or/c string? #f)
(plot-legend-font-face face) → void?
  face : (or/c string? #f)
 = #f
```

```
(plot-legend-font-family) → (or/c font-family/c #f)
(plot-legend-font-family family) → void?
  family : (or/c font-family/c #f)
 = #f
```

The font size (in drawing units), face, and family to prefer for the legend text. If set to `#f`, then the corresponding `plot-font-X` parameter is used.

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

The placement of the legend and the opacity of its background.

```
(plot-legend-layout)
 → (list/c (or/c 'columns 'rows) positive-integer? (or/c 'equal-size 'compact))
(plot-legend-layout layout) → void?
  layout : (list/c (or/c 'columns 'rows) positive-integer? (or/c 'equal-size 'compact))
 = '(columns 1 equal-size)
```

Defines the way in which individual entries are placed in the legend. This is a list of three elements:

- the placement direction (`'columns` or `'rows`)
- the number of columns or rows
- whether all the entries will have the same size (`'equal-size`), or the entries will only occupy the minimum size (`'compact`)

For example, the value `'(columns 1 equal-size)` will place the legend entries vertically from top to bottom and all entries will have the same height. A value of `'(rows 2 'compact)` will place legend entries horizontally on two rows – this type of layout is useful when the legend is placed at the top or bottom of the plot.

Added in version 7.9 of package `plot-gui-lib`.

```
(plot-legend-padding)
```

```
 → (or/c (>=/c 0) (list (>=/c 0) (>=/c 0) (>=/c 0) (>=/c 0)))
(plot-legend-padding padding) → void?
  padding : (or/c (>=/c 0) (list (>=/c 0) (>=/c 0) (>=/c 0) (>=/c 0)))
 = 0
```

The amount of space to add between the legend entries and the border drawn around the legend. The parameter can be specified as a single value, which applies to all sides, or as a list of four separate values for the left, right, top, and bottom sides of the legend.

One example use for this parameter is to avoid clipping thick lines used in legend entries, see `plot-line-width` and `line-width`. In such a case, the end of the lines can be drawn outside the border of the legend, a non-zero `plot-legend-padding` value can be used to avoid this situation.

See also `plot-inset` for a similar setting for the entire plot image.

Added in version 8.11 of package `plot-gui-lib`.

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

The length of tick lines, in drawing units.

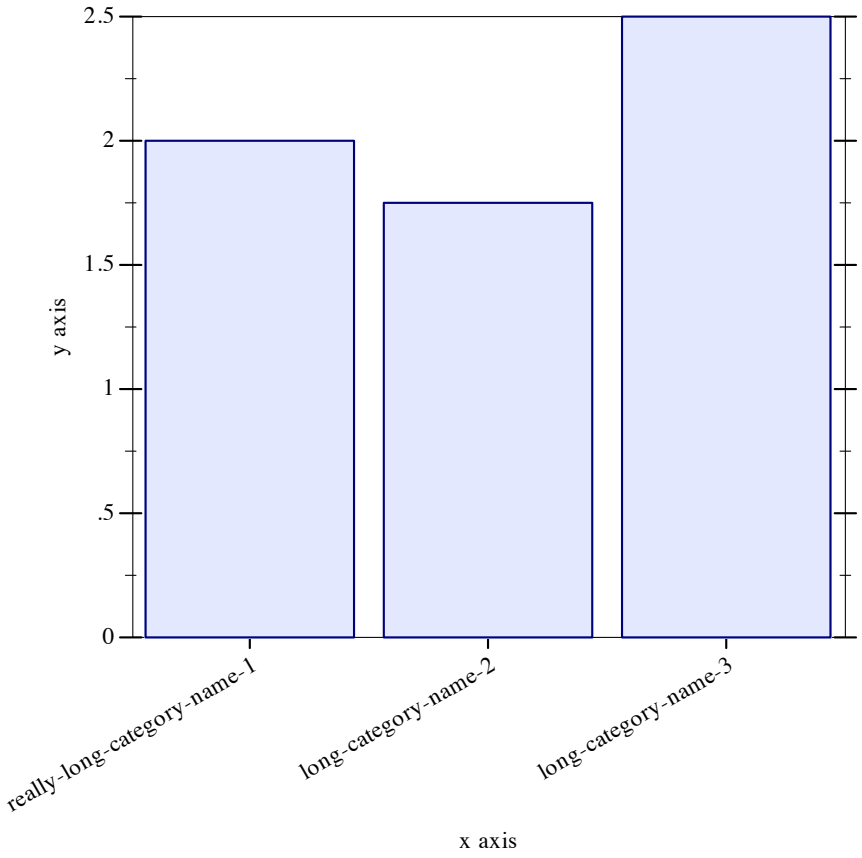
```
(plot-x-tick-label-anchor) → anchor/c
(plot-x-tick-label-anchor anchor) → void?
  anchor : anchor/c
 = 'top
(plot-y-tick-label-anchor) → anchor/c
(plot-y-tick-label-anchor anchor) → void?
  anchor : anchor/c
 = 'right
(plot-x-far-tick-label-anchor) → anchor/c
(plot-x-far-tick-label-anchor anchor) → void?
  anchor : anchor/c
 = 'bottom
(plot-y-far-tick-label-anchor) → anchor/c
(plot-y-far-tick-label-anchor anchor) → void?
  anchor : anchor/c
 = 'left
(plot-x-tick-label-angle) → real?
(plot-x-tick-label-angle angle) → void?
  angle : real?
 = 0
```

```
(plot-y-tick-label-angle) → real?
(plot-y-tick-label-angle angle) → void?
  angle : real?
 = 0
(plot-x-far-tick-label-angle) → real?
(plot-x-far-tick-label-angle angle) → void?
  angle : real?
 = 0
(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? draw?) → void?
  draw? : boolean?
 = #t
(plot-y-axis?) → boolean?
(plot-y-axis? draw?) → void?
  draw? : boolean?
 = #t
(plot-z-axis?) → boolean?
(plot-z-axis? draw?) → void?
  draw? : boolean?
 = #t
(plot-x-far-axis?) → boolean?
(plot-x-far-axis? draw?) → void?
  draw? : boolean?
 = #t
```

```
(plot-y-far-axis?) → boolean?
(plot-y-far-axis? draw?) → void?
  draw? : boolean?
 = #t
(plot-z-far-axis?) → boolean?
(plot-z-far-axis? draw?) → void?
  draw? : boolean?
 = #t
```

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

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

```
(plot-x-tick-labels?) → boolean?
(plot-x-tick-labels? draw?) → void?
  draw? : boolean?
 = #t
(plot-y-tick-labels?) → boolean?
(plot-y-tick-labels? draw?) → void?
  draw? : boolean?
 = #t
(plot-z-tick-labels?) → boolean?
(plot-z-tick-labels? draw?) → void?
  draw? : boolean?
 = #t
(plot-x-far-tick-labels?) → boolean?
(plot-x-far-tick-labels? draw?) → void?
  draw? : boolean?
(plot-y-far-tick-labels?) → boolean?
(plot-y-far-tick-labels? draw?) → void?
  draw? : boolean?
(plot-z-far-tick-labels?) → boolean?
(plot-z-far-tick-labels? draw?) → void?
  draw? : boolean?
```

When any of these is `#f`, the corresponding labels for the ticks on the axis are not drawn. These parameters work together with the parameters like `plot-x-axis?` that control the drawing of the axes; i.e. tick labels won't be drawn unless the axis itself is drawn.

```
(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))
```

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? draw?) → void?
  draw? : boolean?
 = #t
```

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

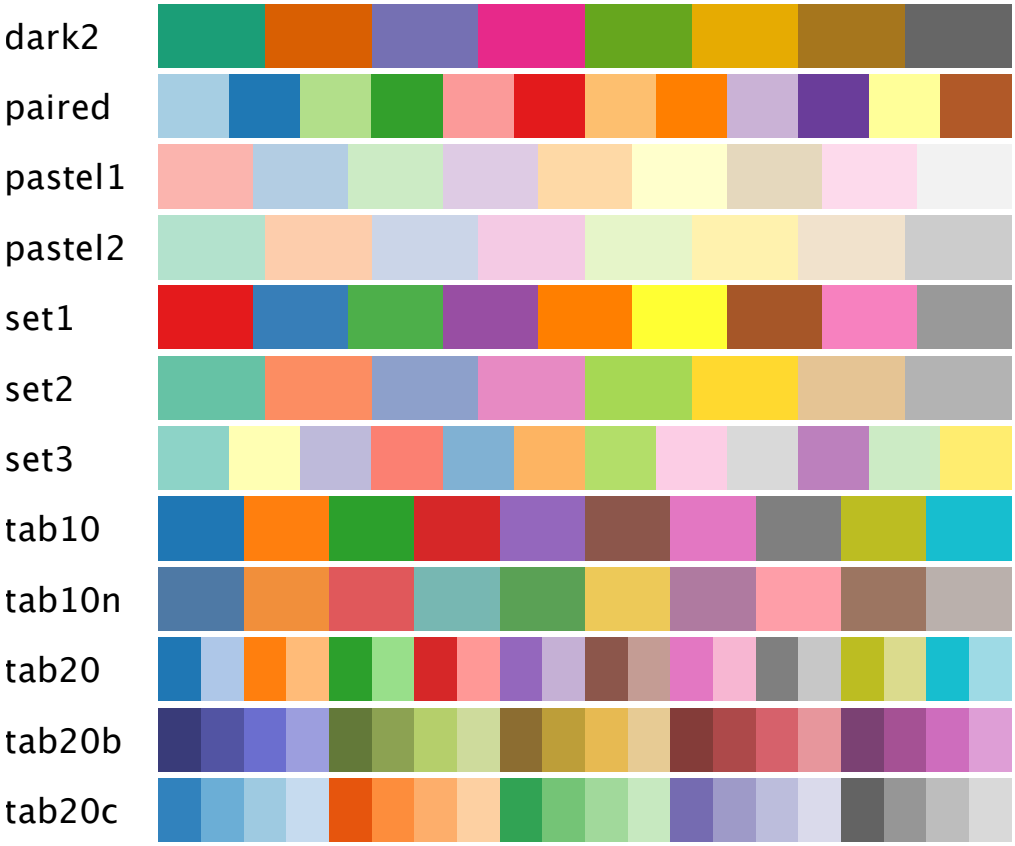
```
(plot-pen-color-map) → (or/c symbol? #f)
(plot-pen-color-map name) → void?
  name : (or/c symbol? #f)
 = #f
(plot-brush-color-map) → (or/c symbol? #f)
(plot-brush-color-map name) → void?
  name : (or/c symbol? #f)
 = #f
```

Specify the color maps to be used by ->pen-color and ->brush-color respectively, for converting integer values into RGB triplets, or when integer values are used with the #:color keyword of various plot renderers. You can determine the list of available color map names using color-map-names.

If *name* is not a valid color map name, the internal color map will be used, this is the same as specifying #f.

When the color map value is set to #f, internal color maps will be used, one for pen and one for brush colors. The internal color map used for pen colors has darker and more saturated colors than the one used for brush colors. These colors are chosen for good pairwise contrast, especially between neighbors and they repeat starting with 128.

The color maps available by default are shown below and additional ones can be added using register-color-map:




Added in version 7.3 of package `plot-gui-lib`.

## 8.4 Lines

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

The number of points to sample when approximating a line. Used as a default keyword argument in `function`, `inverse`, `parametric`, `polar`, `density`, `function-interval`, `inverse-interval`, `parametric-interval`, `polar-interval`, `area-histogram` and `parametric3d`.

```
(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-cap) → plot-pen-cap/c
(line-cap cap) → void?
  cap : plot-pen-cap/c
 = 'round
(line-alpha) → (real-in 0 1)
(line-alpha alpha) → void?
  alpha : (real-in 0 1)
 = 1
```

The pen color, pen width, pen style, pen cap and opacity of lines in plots.

Except for `line-cap`, all other parameters are used as default keyword arguments of `function`, `inverse`, `lines`, `parametric`, `polar`, `density`, `isoline`, `lines3d`, `parametric3d` and `isoline3d`.

The `line-cap` parameter applies to lines drawn by renderers in a plot. See also `plot-line-cap`.

Added in version 8.10 of package `plot-gui-lib`.

## 8.5 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
```

The brush color/style, lower line pen color/width/style, upper line pen color/width/style, and opacity of interval plots. Used as default keyword arguments of `function-interval`, `inverse-interval`, `lines-interval`, `parametric-interval` and `polar-interval`.

## 8.6 Points and Point Labels

```
(point-sym) → point-sym/c
(point-sym sym) → void?
  sym : point-sym/c
 = 'circle
(point-size) → (>=/c 0)
(point-size size) → void?
  size : (>=/c 0)
 = 6
(point-alpha) → (real-in 0 1)
(point-alpha alpha) → void?
  alpha : (real-in 0 1)
 = 1
```

The symbol, and its size and opacity, used in point plots. Used as default keyword arguments of `points` and `points3d`.

```
(point-x-jitter) → (>=/c 0)
(point-x-jitter x-jitter) → void?
  x-jitter : (>=/c 0)
 = 0
(point-y-jitter) → (>=/c 0)
(point-y-jitter y-jitter) → void?
  y-jitter : (>=/c 0)
 = 0
(point-z-jitter) → (>=/c 0)
(point-z-jitter z-jitter) → void?
  z-jitter : (>=/c 0)
 = 0
```

When any of *x-jitter*, *y-jitter*, or *z-jitter* are non-zero, `points` and `points3d` will produce points randomly translated from their original position along the x, y, or z axis, respectively. For instance, if each parameter is set to 0.5, then `points '(0 0)` will produce a random point in a square of area 1 centered at `'(0 0)`. Likewise `points3d` will make a random point within a unit cube centered at `'(0 0 0)`.

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

The color and line width of symbols used in point plots and labeled points. Used as default keyword arguments of `points` and `points3d`, as well as in `point-label`, `function-label`, `inverse-label`, `parametric-label`, `polar-label` and `point-label3d`.

```
(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
```

Point label anchor, angle, and opacity, and the size of points next to labels. Used as default keyword arguments of `point-label`, `function-label`, `inverse-label`, `parametric-label`, `polar-label` and `point-label3d`.

## 8.7 Vector Fields & Arrows

```
(vector-field-samples) → exact-positive-integer?
(vector-field-samples n) → void?
  n : exact-positive-integer?
 = 20
(vector-field3d-samples) → exact-positive-integer?
(vector-field3d-samples n) → void?
  n : exact-positive-integer?
 = 9
```

The default number of samples `vector-field` and `vector-field3d` take, respectively.

```
(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
```

The default pen color, pen width, pen style, scaling factor, and opacity used by vector-field and vector-field3d.

```
(arrows-color) → plot-color/c
(arrows-color color) → void?
  color : plot-color/c
 = 1
(arrows-line-width) → (>=/c 0)
(arrows-line-width width) → void?
  width : (>=/c 0)
 = 2/3
(arrows-line-style) → plot-pen-style/c
(arrows-line-style style) → void?
  style : plot-pen-style/c
 = 'solid
(arrows-alpha) → (real-in 0 1)
(arrows-alpha alpha) → void?
  alpha : (real-in 0 1)
 = 1
```

The default pen color, pen width, pen style, and opacity used by arrows and arrows3d.

Added in version 7.9 of package plot-gui-lib.

```
(arrow-head-size-or-scale)
 → (or/c (>=/c 0) (list/c '= (>=/c 0)))
(arrow-head-size-or-scale size) → void?
  size : (or/c (>=/c 0) (list/c '= (>=/c 0)))
 = 2/5
(arrow-head-angle) → (>=/c 0)
(arrow-head-angle alpha) → void?
  alpha : (>=/c 0)
 = (/ pi 6)
```

The default size and angle of the arrow head in vector-field, vector-field3d, arrows and arrows3d. When the arrow-head-size-or-scale is a number, it is interpreted as a proportion of the arrow length , and will be bigger for longer arrows. When it is in the form (list '= size), it is interpreted as the size of the arrow head in drawing units (pixels).

Added in version 7.9 of package plot-gui-lib.

## 8.8 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 pen-width) → void?
  pen-width : (>=/c 0)
 = 1
(error-bar-line-style) → plot-pen-style/c
(error-bar-line-style pen-style) → void?
  pen-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
```

The default width, pen color/width/style, and opacity used by `error-bars`.

## 8.9 Candlesticks

```
(candlestick-width) → (>=/c 0)
(candlestick-width width) → void?
  width : (>=/c 0)
 = 1
(candlestick-up-color) → plot-color/c
(candlestick-up-color color) → void?
  color : plot-color/c
 = 2
(candlestick-down-color) → plot-color/c
(candlestick-down-color color) → void?
  color : plot-color/c
 = 1
(candlestick-line-width) → (>=/c 0)
(candlestick-line-width pen-width) → void?
  pen-width : (>=/c 0)
 = 1
(candlestick-line-style) → plot-pen-style/c
(candlestick-line-style pen-style) → void?
  pen-style : plot-pen-style/c
 = 'solid
```

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

The default width, pen color/width/style, and opacity used by `candlesticks`. Both the up (a candle whose open value is lower than its close value) color and the down (a candle whose open value is higher than its close value) color can be specified independently. The width parameter will be important to specify if your x-axis is in units like days, weeks, or months. Because dates are actually represented as seconds from an epoch, your width should take that into consideration. For example, a width of 86400 may be useful for x-axis values in days as there are 86400 seconds in a day. This candle will be exactly one day in width.

## 8.10 Color fields

```
(color-field-samples) → exact-positive-integer?
(color-field-samples n) → void?
  n : exact-positive-integer?
 = 20
(color-field-alpha) → (real-in 0 1)
(color-field-alpha alpha) → void?
  alpha : (real-in 0 1)
 = 1
```

The default sample rate and opacity used by `color-field`.

Added in version 7.9 of package `plot-gui-lib`.

## 8.11 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))
```

The default values of the parameters `contour-colors` and `contour-interval-colors`, respectively.

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

The number of samples taken in 2D contour plots. Used as a defaut keyword argument in `isoline`, `contours` and `contour-intervals`.

```
(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)
```

The number, pen colors, pen widths, and pen styles of **lines** in contour plots. Used as default keyword arguments of `contours`, `contour-intervals`, `contours3d`, and `contour-intervals3d`.

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

The opacities of **lines** in contour plots. Used as a default keyword argument in `contours` and `contours3d`.

```
(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)
```

The brush colors, brush styles, and opacities of **intervals** in contour plots. Used as default keyword arguments of `contour-intervals` and `contour-intervals3d`.

## 8.12 Contour Surfaces

```
(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)
```

The pen colors, widths, and styles of the sampling grid, where it intersects contour intervals. Used as default keyword arguments of `contour-intervals3d`.

## 8.13 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 pen-color) → void?
  pen-color : plot-color/c
 = 3
```

```
(rectangle-line-width) → (>=/c 0)
(rectangle-line-width pen-width) → void?
  pen-width : (>=/c 0)
 = 1
(rectangle3d-line-width) → (>=/c 0)
(rectangle3d-line-width pen-width) → void?
  pen-width : (>=/c 0)
 = 1/3
(rectangle-line-style) → plot-pen-style/c
(rectangle-line-style pen-style) → void?
  pen-style : plot-pen-style/c
 = 'solid
(rectangle-alpha) → (real-in 0 1)
(rectangle-alpha alpha) → void?
  alpha : (real-in 0 1)
 = 1
```

The brush color/style of faces, pen color/width/style of edges, and opacity of recangles. Used as default keyword arguments of `rectangles`, `area-histogram`, `discrete-histogram`, `rectangles3d` and `discrete-histogram3d`.

The default pen width of 3D rectangle edges is narrower for aesthetic reasons.

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

The gap between histogram bars, as a percentage of bar width. Used as a default keyword argument of `discrete-histogram`, `stacked-histogram`, `discrete-histogram3d` and `stacked-histogram3d`.

```
(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
```

Distance on the *x* axis between histogram bars, and whether to draw histograms horizontally. Used as default keyword arguments of `discrete-histogram` and `stacked-histogram`.

```
(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-styles) → (plot-brush-styles/c nat/c)
(stacked-histogram-styles styles) → void?
  styles : (plot-brush-styles/c nat/c)
 = '(solid)
(stacked-histogram-line-colors) → (plot-colors/c nat/c)
(stacked-histogram-line-colors pen-colors) → void?
  pen-colors : (plot-colors/c nat/c)
 = (stacked-histogram-colors)
(stacked-histogram-line-widths) → (pen-widths/c nat/c)
(stacked-histogram-line-widths pen-widths) → void?
  pen-widths : (pen-widths/c nat/c)
 = '(1)
(stacked-histogram-line-styles) → (plot-pen-styles/c nat/c)
(stacked-histogram-line-styles pen-styles) → void?
  pen-styles : (plot-pen-styles/c nat/c)
 = '(solid)
(stacked-histogram-alphas) → (alphas/c nat/c)
(stacked-histogram-alphas alphas) → void?
  alphas : (alphas/c nat/c)
 = '(1)
```

Stacked histogram brush colors/styles, pen colors/widths/styles, and opacities. Used as default keyword arguments of `stacked-histogram` and `stacked-histogram3d`.

## 8.14 Non-Border Axes

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

Default values for keyword arguments of `x-axis`, `y-axis` and `axes`.

```
(polar-axes-number) → exact-nonnegative-integer?
(polar-axes-number n) → void?
  n : exact-nonnegative-integer?
 = 12
(polar-axes-ticks?) → boolean?
(polar-axes-ticks? ticks?) → void?
  ticks? : boolean?
 = #t
(polar-axes-labels?) → boolean?
(polar-axes-labels? labels?) → void?
  labels? : boolean?
 = #t
(polar-axes-alpha) → (real-in 0 1)
(polar-axes-alpha alpha) → void?
  alpha : (real-in 0 1)
 = 1/2
```

Number of polar axes, whether radius ticks (i.e. lines) are drawn, whether labels are drawn, and opacity. Used as default keyword arguments of `polar-axes`.

## 8.15 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 pen-color) → void?
  pen-color : plot-color/c
 = 0
(surface-line-width) → (>=/c 0)
(surface-line-width pen-width) → void?
  pen-width : (>=/c 0)
 = 1/3
(surface-line-style) → plot-pen-style/c
(surface-line-style pen-style) → void?
  pen-style : plot-pen-style/c
 = 'solid
(surface-alpha) → (real-in 0 1)
(surface-alpha alpha) → void?
  alpha : (real-in 0 1)
 = 1
```

Surface brush color/style, pen color/width/style of the sampling grid where it intersects the surface, and opacity. Used as default keyword arguments of `surface3d`, `polar3d` and `isosurface3d`.

```
(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))
```

The default values of the parameters `isosurface-colors` and `isosurface-line-colors`, respectively.

```
(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 pen-colors) → void?
  pen-colors : (plot-colors/c (listof real?))
 = default-isosurface-line-colors
(isosurface-line-widths) → (pen-widths/c (listof real?))
(isosurface-line-widths pen-widths) → void?
  pen-widths : (pen-widths/c (listof real?))
 = '(1/3)
(isosurface-line-styles) → (plot-pen-styles/c (listof real?))
(isosurface-line-styles pen-styles) → void?
  pen-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)
```

The number, brush colors/styles, pen colors/widths/styles, grid color/widths/styles, and opacities of nested isosurfaces. Used as default keyword arguments of `isosurfaces3d`.

# 9 Plot Contracts

```
(require plot/utils)
```
package: plot-lib

## 9.1 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 §4 "3D Renderers" for functions that construct them.

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

Returns `#t` if *value* is a nonrenderer. See §5 "Nonrenderers" for functions that construct them.

```
(treeof elem-contract) → contract?
  elem-contract : contract?
```

Identifies values that meet the contract *elem-contract*, lists of such values, lists of lists, and so on.

## 9.2 Appearance Argument Contracts

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

The contract for `anchor` arguments and parameters.

The `'auto` anchor will place labels so they are visible on the plot area. This anchor type is useful for `point-label` and similar renderers where the labeled point might be at the edge of the plot area and the user does not wish to calculate the exact anchor for the label.

The 'auto anchor will choose one of the 'bottom-left, 'bottom-right, 'top-left or 'top-right placements, in that order, and will use the first one that would result in the label being completely visible.

The 'auto anchor is only valid for placement of text labels, for all other use cases, the 'auto anchor is always the same as 'bottom-left.

```
legend-anchor/c : contract?
 = (or/c anchor/c
         (one-of/c 'outside-top-left    'outside-top    'outside-top-right
                   'outside-left-top    'outside-left   'outside-left-bottom
                   'outside-right-top   'outside-right  'outside-right-bottom
                   'outside-bottom-left 'outside-bottom 'outside-bottom-right
                   'outside-global-top
                   'no-legend))
```

The contract for the plot-legend-anchor parameter and the #:legend-anchor parameters for the various plot procedures.

When legend-anchor is one of the symbols from anchor/c, the legend will be placed inside the plot area.

legend-anchor/c values which start with "outside" will place the legend outside the plot area, for 2D plots the legend will be aligned with the plot area, while for 3D plots the legend will be relative to the overall plot-width and plot-height.

The 'outside-global-top value will place the legend above the plot-area, centered on the complete plot-width. For 3D plots there is no difference between this value and 'outside-top.

The value 'no-legend, will omit the legend from the plot, the legend will also be omitted if none of the renderers have a #:label specified, regardless of the value used for #:legend-anchor.

The value 'auto, will place the legend in the top-left corner of the plot area, this is not usefull for plot legends, this anchor value is used for renderers such as point-label.

Added in version 7.9 of package plot-lib.

```
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-pen-cap/c : contract? = (one-of/c 'round 'projecting 'butt)
```

The contract for caps, or line endings, for lines drawn on the plot. Used by the `plot-line-cap` and `line-cap` parameters.

Added in version 8.10 of package `plot-lib`.

```
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 : flat-contract?
```

Identifies legal font family values. The same as `font-family/c` from `racket/draw`.

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

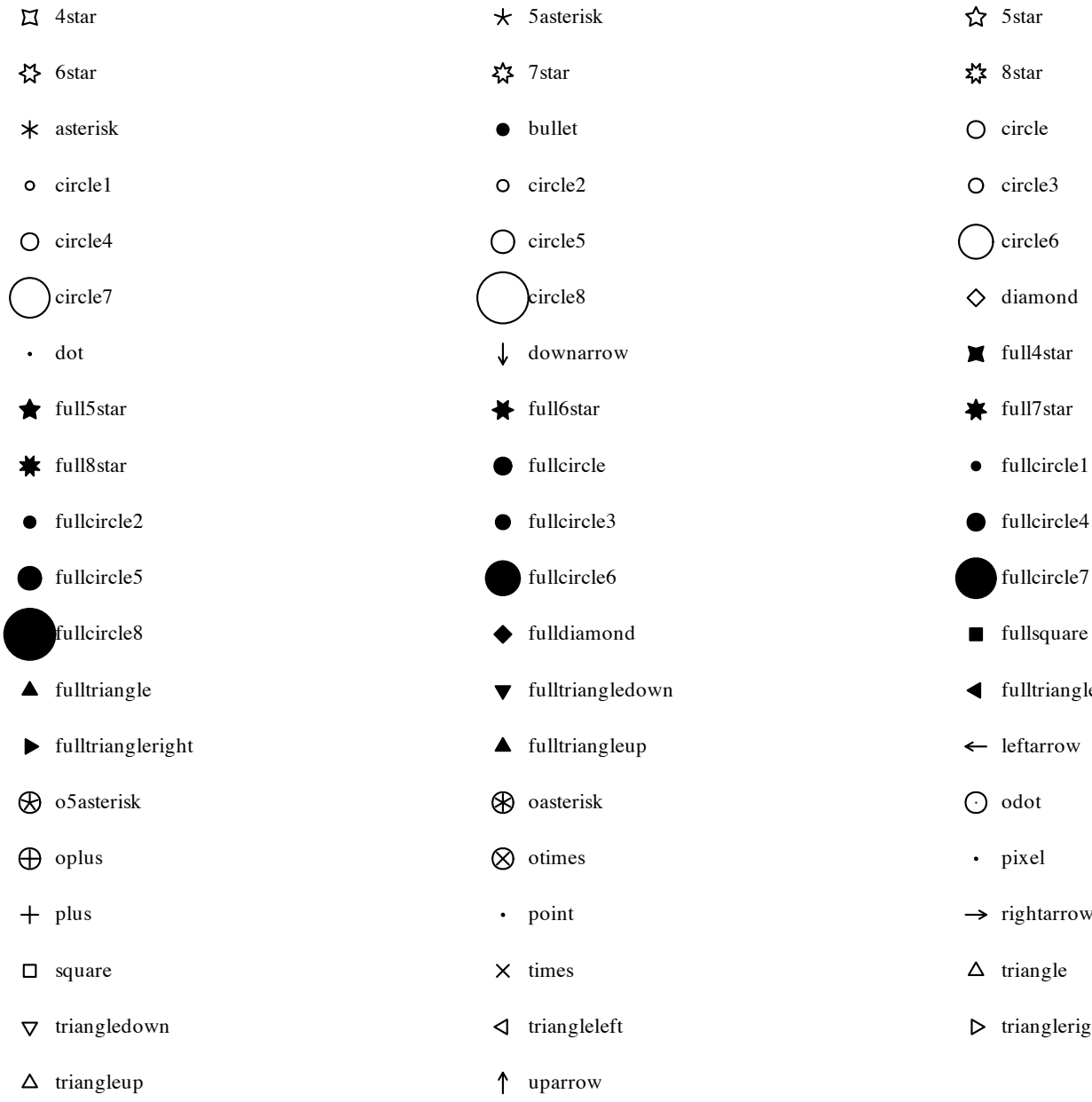
The contract for the `#:sym` arguments in `points` and `points3d`, and the parameter `point-sym`.

Characters and strings will render that character or string for each point on the plot, one of the symbols in `known-point-symbols` will render the corresponding symbol, while an integer value represents an index into the `known-point-symbols` list, this can be used to automatically generate distinct symbols for different point renderers by incrementing a number.

```
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      'none)
```

A list containing the symbols that are valid `points` symbols, the rendering of each symbol is shown below.




```
plot-file-format/c : contract?
 = (or/c 'auto 'png 'jpeg 'xmb 'xpm 'bmp 'ps 'pdf 'svg)
```

A contract for an argument that describes an image file format.

## 9.3 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*.

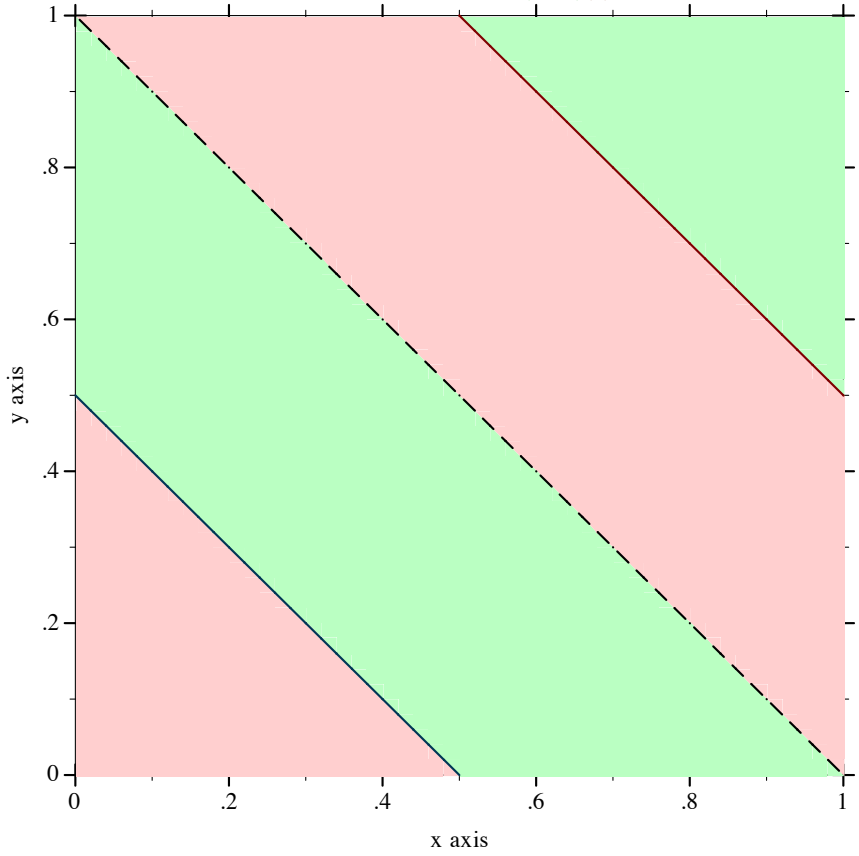
This is used inside many renderer-producing `plot` functions to convert `maybe-function/c` values to lists of appearance values.

```
(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)))
```

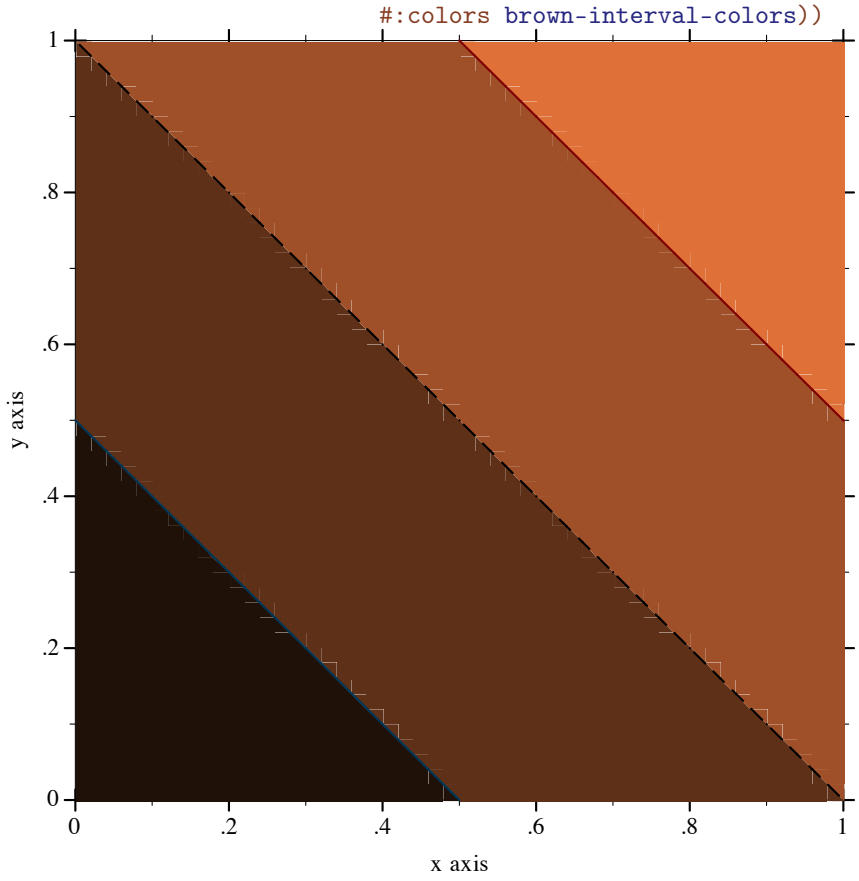



This is equivalent to sending `(λ _ '(1 2))`.

The next example is more sophisticated: it sends a *function*-valued `(plot-colors/c ivl?)` 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))
```




```
(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-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.

```
(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? pict? #f)))
```

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

# 10 Porting From Plot <= 5.1.3

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

## 10.1 Replacing Deprecated Functions

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

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

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

## 10.2 Ensuring That Plots Have Bounds

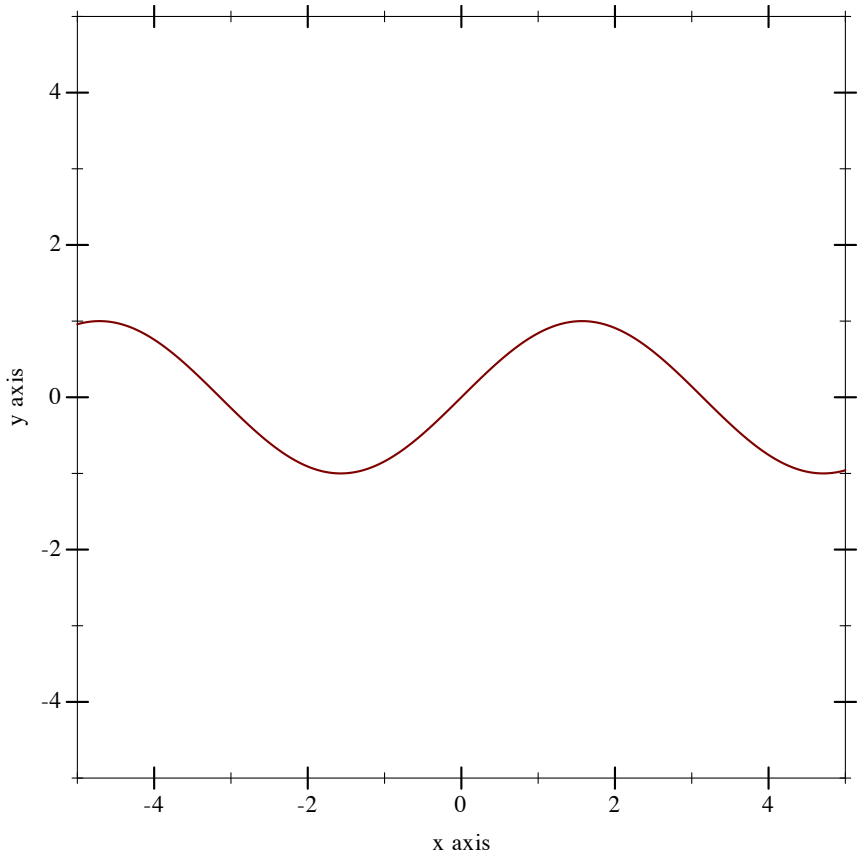
The safest way to ensure that `plot` can determine bounds for the plot area is to add `#:x-min -5 #:x-max 5 #:y-min -5 #:y-max 5` to every call to `plot`. Similarly, add `#:x-min -5 #:x-max 5 #:y-min -5 #:y-max 5 #:z-min -5 #:z-max 5` to every call to `plot3d`.

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

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

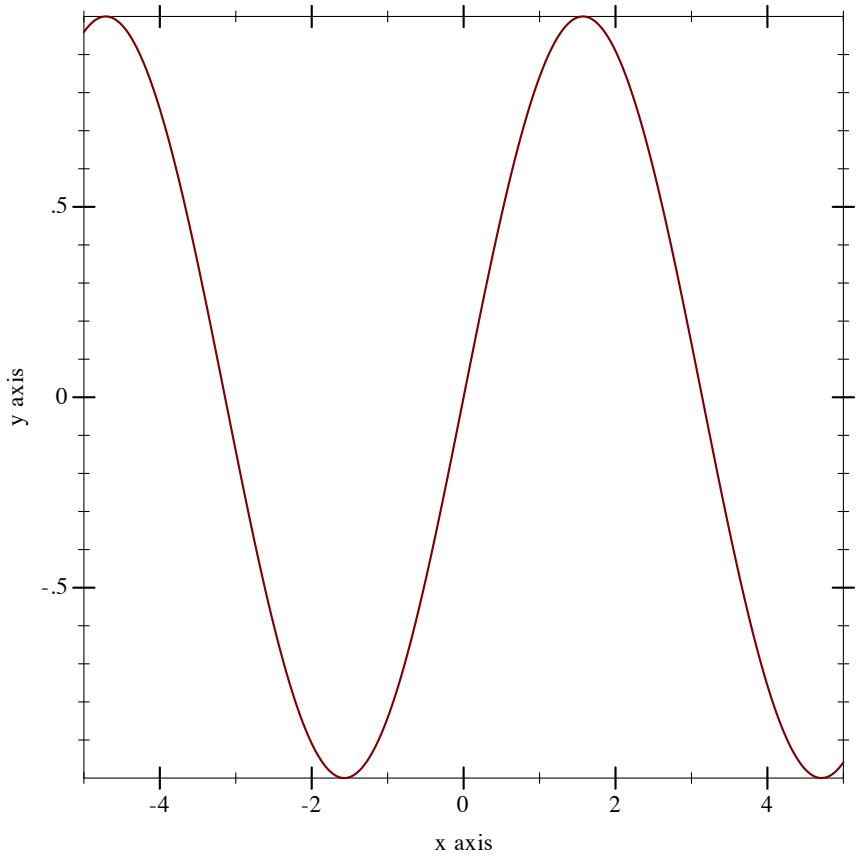
You could either change it to

```
> (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.

## 10.3 Changing Keyword Arguments

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

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

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

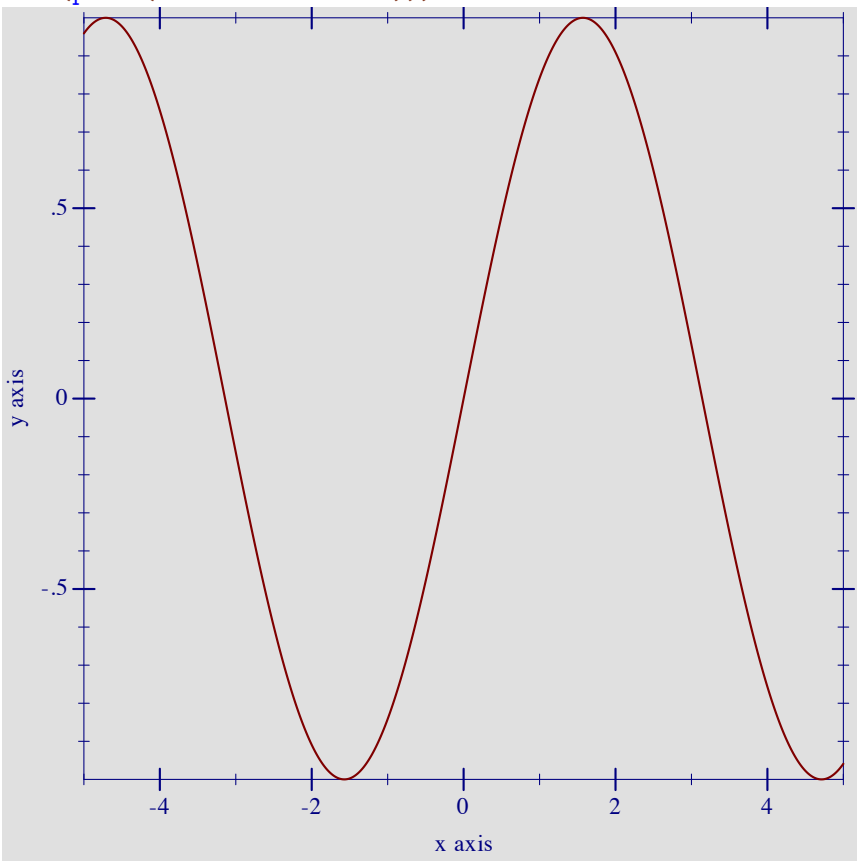
Replace every `#:style 'normalized` with `#:scale 'normalized`.

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

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

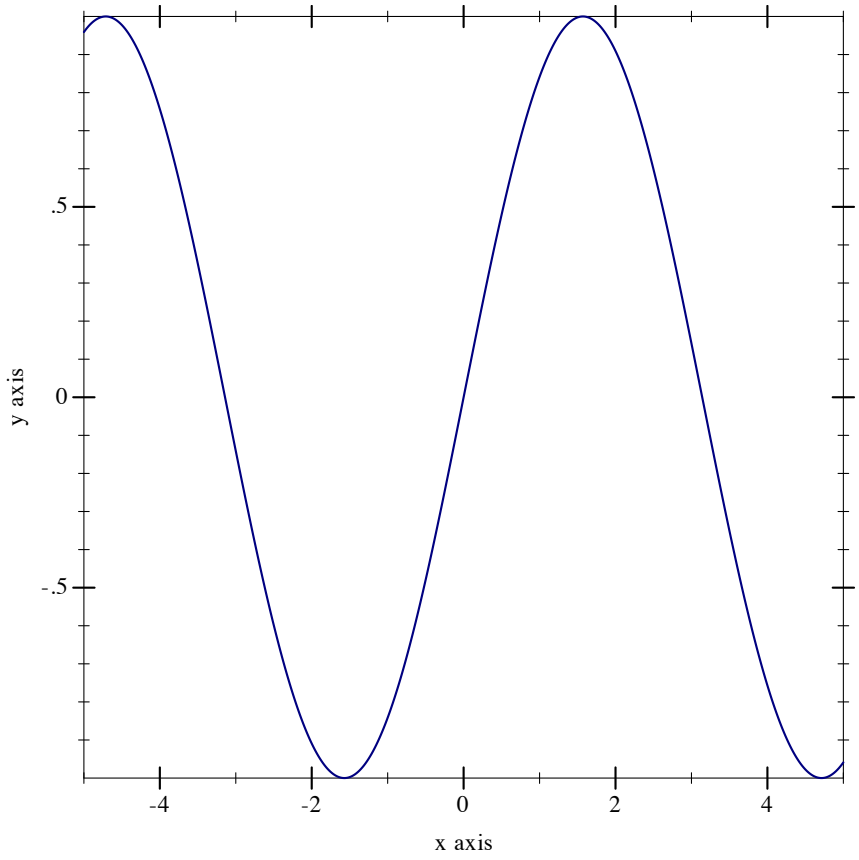
`#:bgcolor '(224 224 224))`, change it to

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




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

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




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

## 10.4 Fixing Broken Calls to `points`

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

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

## 10.5 Replacing Uses of `plot-extend`

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

The rewritten Plot will eventually have a similar extension mechanism.

## 10.6 Deprecated Functions

`(require plot)` package: `plot-gui-lib`

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.

# 11 Legacy Typed Interface

```
(require plot/typed)
```
package: `plot-gui-lib`

```
(require plot/typed/utils)
 (require plot/typed/no-gui)
 (require plot/typed/bitmap)
 (require plot/typed/pict)
```
package: `plot-lib`

**Do not use these modules in new programs.** They are likely to disappear in a (distant) future release. Use `plot`, `plot/utils`, `plot/no-gui`, `plot/bitmap` and `plot/pict` instead.

Plot versions 6.1.1 and earlier were written in untyped Racket, and exposed a typed interface through these modules. Now that Plot is written in Typed Racket, a separate typed interface is no longer necessary. However, the above modules are still available for backwards compatibility.

# 12 Compatibility Module

`(require plot/compat)` package: `plot-compat`

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

**Do not use this module in new programs.** It is likely to disappear in a near future release.

**Do not try to use both `plot` and `plot/compat` in the same program.** The new features in Plot 5.2 and later require 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 errors.

## 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 an `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-6
```

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-6
```

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.