AN INTRODUCTION TO R

DEEPAYAN SARKAR

In this tutorial session, we will learn about one of the strongest features of R, its graphics facilities. There are two distinct graphics systems built into R, referred to as *traditional* and *grid* graphics. Other packages provide further functionality. We will first discuss traditional graphics.

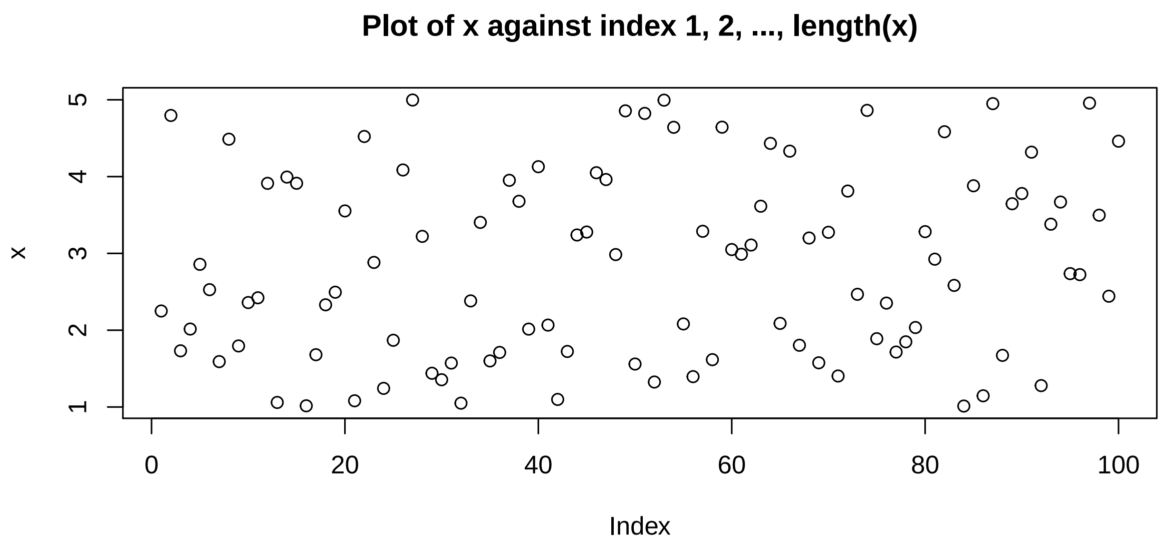
Traditional R graphics

R's graphics system is actually fairly complicated, with many features that are rarely used. Instead of going into details immediately, we will learn about R graphics by looking at some examples.

Plotting functions in R can be divided into three basic groups: *High-level* plotting functions create a new plot on the graphics device, possibly with axes, labels, titles and so on. *Low-level* plotting functions add more information to an existing plot, such as extra points, lines and labels. *Interactive* graphics functions allow you interactively add information to, or extract information from, an existing plot, using a pointing device such as a mouse. In addition, R maintains a list of graphical parameters that affect the result of various plot functions.

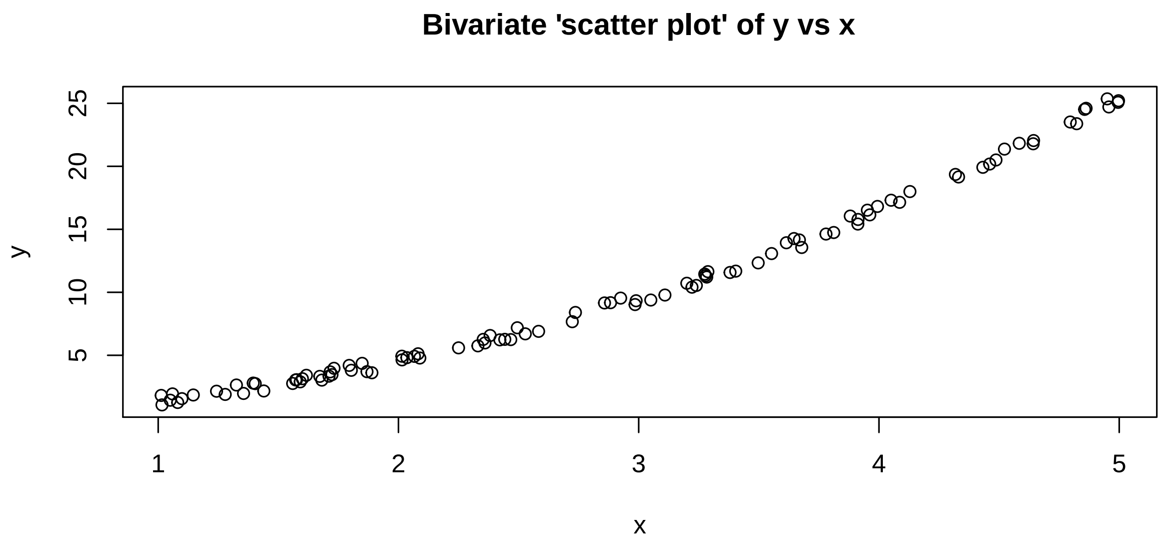
The `plot()` function. The Generic `plot()` function can deal with the task of plotting several types of R objects. The most common use is to plot single or paired numeric vectors.

```
> x <- runif(100, min = 1, max = 5)
> y <- x^2 + runif(100)
> plot(x, main = "Plot of x against index 1, 2, ..., length(x)")
```



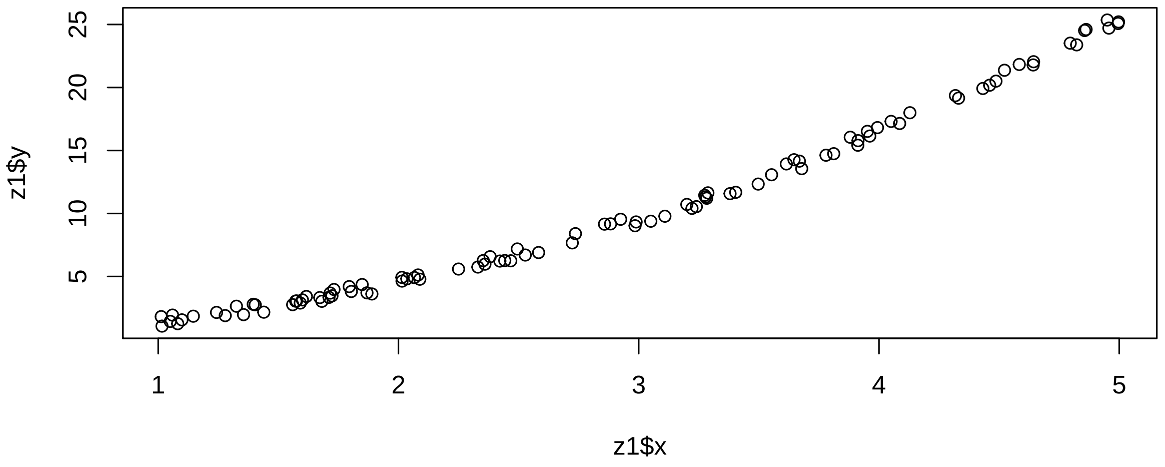
Date: October 2011.

```
> plot(x, y, main = "Bivariate 'scatter plot' of y vs x")
```



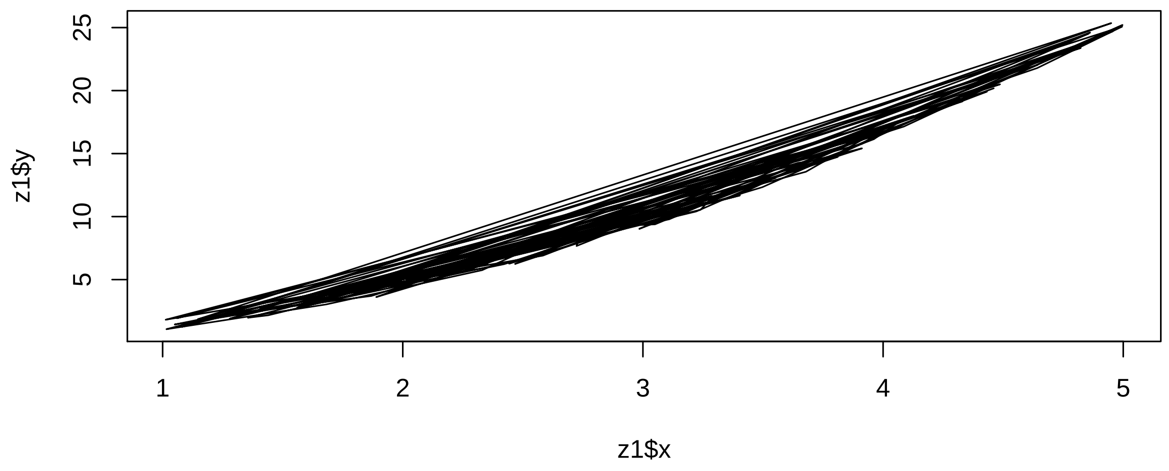
We can also create a single list object with components x and y, and plot it directly. This technique is used in many R functions.

```
> z1 <- list(x = x, y = y)
> plot(z1)
```



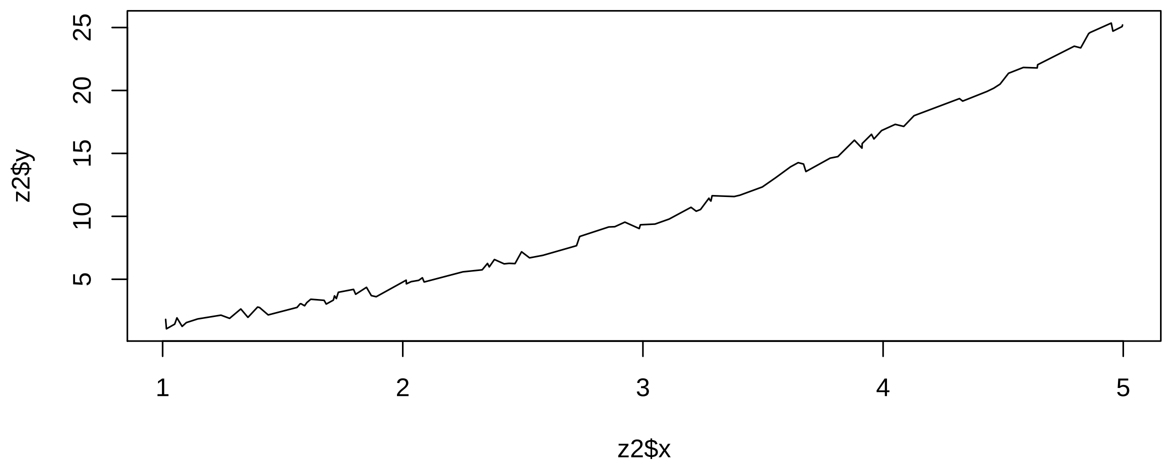
Many variations on the basic plot are possible.

```
> plot(z1, type = "l")
```

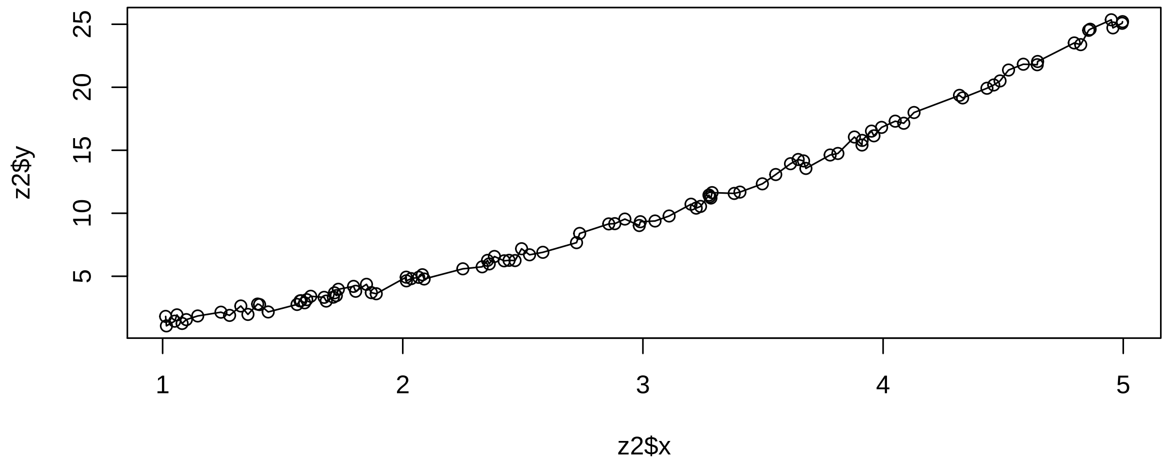


As the points are not in any particular order, this is all jumbled up. To plot the points in increasing order of `x`, we can use

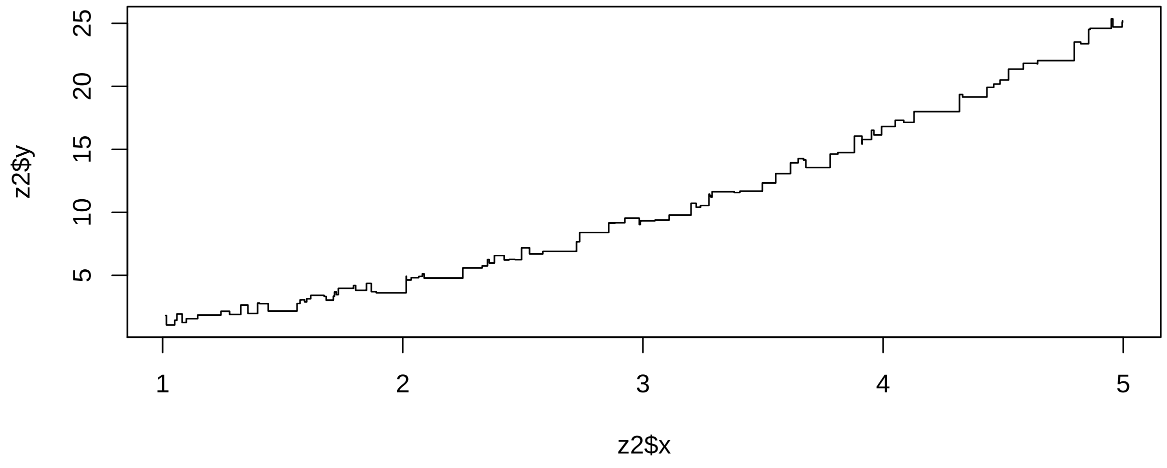
```
> ord <- order(x)
> z2 <- list(x = x[ord], y = y[ord])
> plot(z2, type = "l") # lines
```



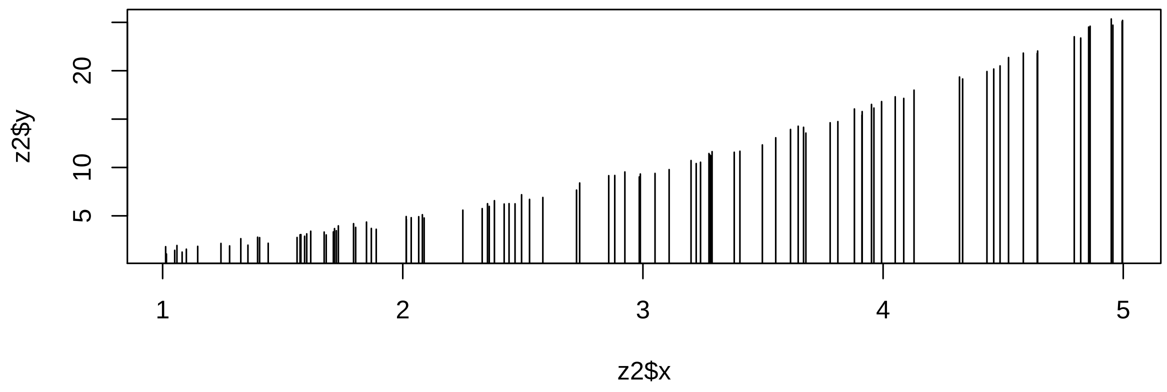
```
> plot(z2, type = "o") # points and lines overlayed
```



```
> plot(z2, type = "s") # steps
```

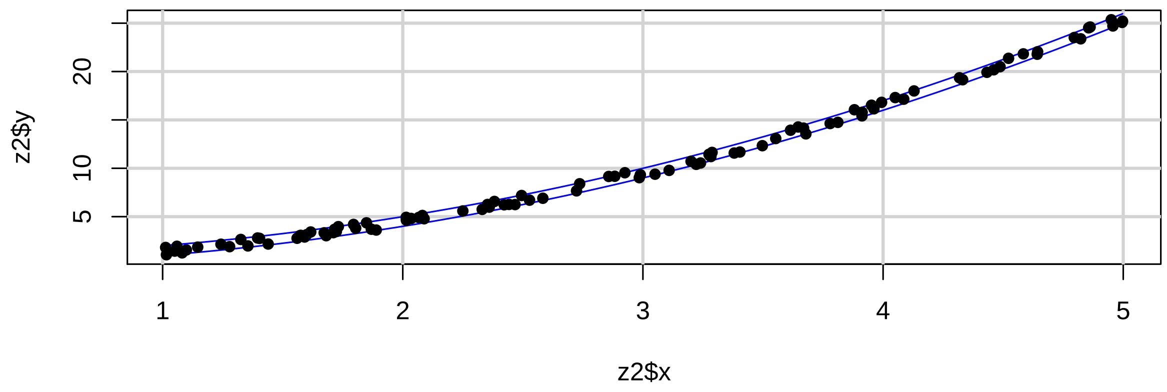


```
> plot(z2, type = "h") # histogram-like
```



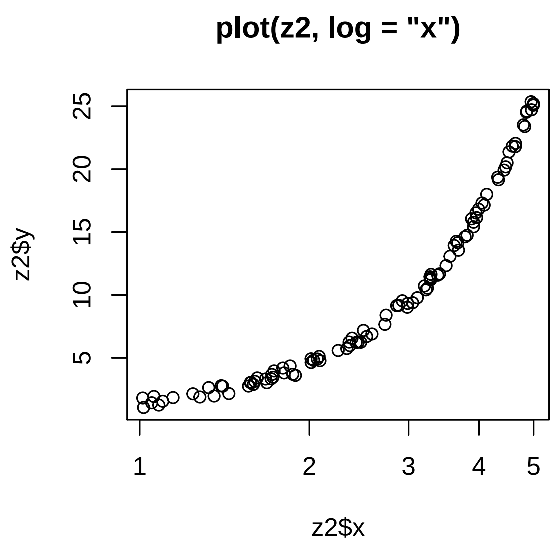
A special value of `type` is `type="n"` which sets up the appropriate axis limits but does not actually plot anything. This is useful for subsequently adding pieces to the plot. In the following example, we add a reference grid and a curve representing the dependence of `y` on `x` *before* plotting the points.

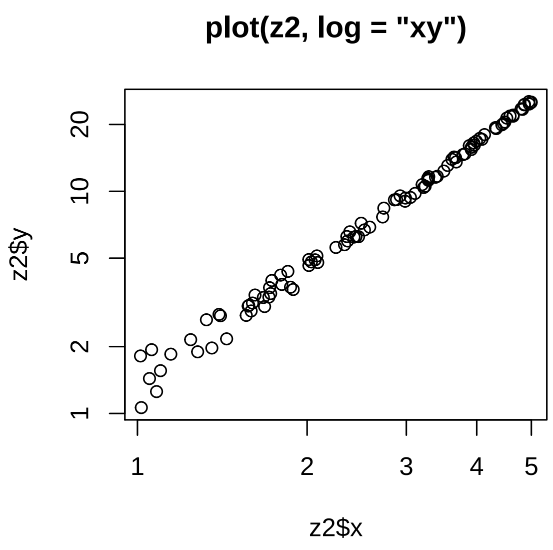
```
> plot(z2, type = "n")
> grid(lty = 1, lwd = 2)
> curve(x^2, col = "blue", add = TRUE)
> curve(x^2 + 1, col = "blue", add = TRUE)
> points(z2, pch = 16)
```



Axis limits are automatically determined to encompass the range of all the data, but can be overridden by specifying arguments `xlim` and `ylim`.

It is common to use log-scales on one or both axes.





Graphical parameters. There are several standard graphical parameters that can be controlled by extra arguments to plot functions. These include color (argument `col`), plotting character (`pch`), size factor or character expansion (`cex`), line type (`lty`), and line width (`lwd`). These parameters usually operate in a vectorized manner. Examples:

```
> plot(z2, type = "o", col = 'red', pch = 16, cex = 2)
> plot(z2, col = c('red', 'blue'), pch = "+", cex = 2)
```

Exercise 1. *Usage of these graphical parameters are described in these two help pages: `?plot.default` and `?par`. Read them and modify the previous commands to obtain different colors, plotting characters, line types, etc.*

The `?par` help page is quite extensive and contains a lot of information. Skip the irrelevant entries on the first reading, but you should read it in more detail later to learn more about the available features.

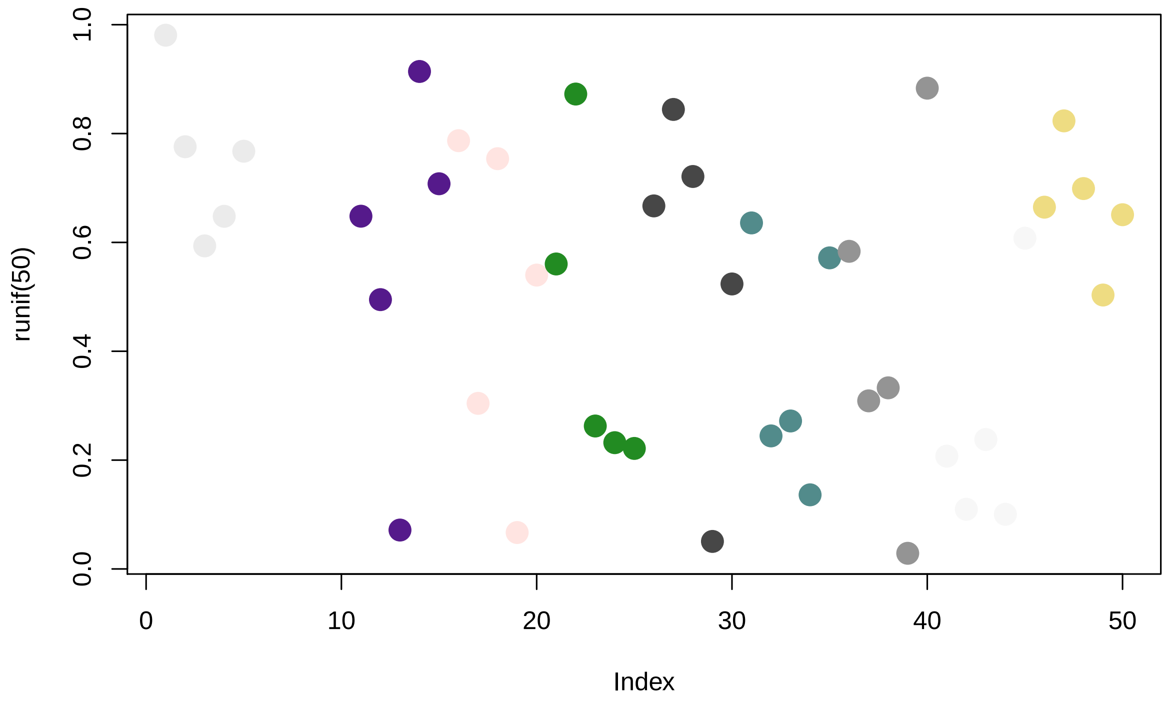
Colors. There are several ways to specify color in R. They can be specified as a descriptive name (`"red"`, `"grey"`, etc. The full list of available names is produced by

```
> colors()
```

A few of these colors can be set as a "palette" and their elements specified by integer codes.

```
> palette() ## default palette
[1] "black"   "red"     "green3"  "blue"    "cyan"    "magenta" "yellow"
[8] "gray"
```

```
> palette(sample(colors(), 10)) ## change palette
> plot(runif(50), col = rep(1:10, each = 5), pch = 16, cex = 2)
```



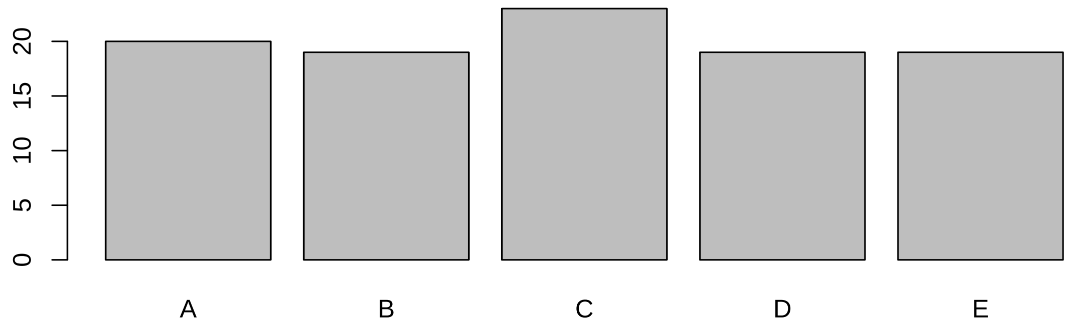
Other functions that can be used to specify colors are `gray()` (for gray-levels) and `rgb()` (arbitrary colors as RGB triplets).

Other useful arguments that almost all high-level plot functions have are `xlab`, `ylab`, `main`, and `sub`. These are all used to annotate the plot by adding labels, titles, and sub-titles. They can also be added after a plot has been created using the `title()` function.

Non-default plot methods. So far, we have only used the default plot method. There are several other plot types which can be produced by the `plot()` function, depending on what object is being plotted.

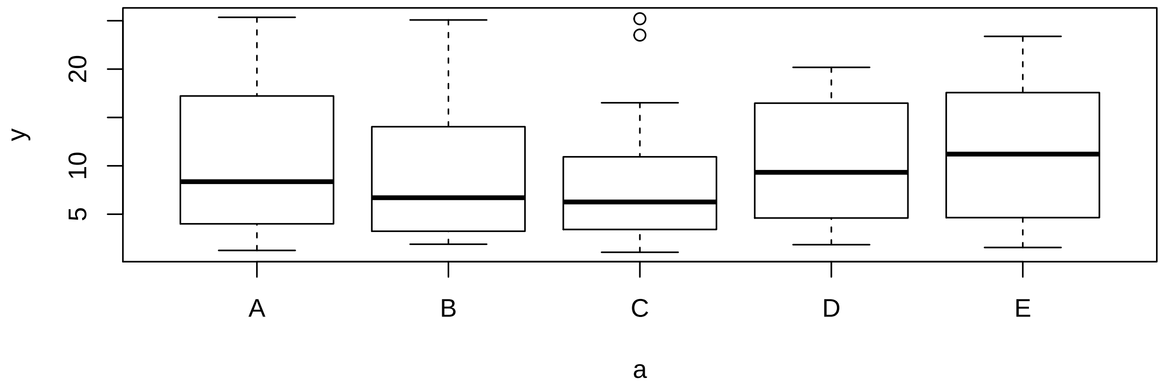
```
> ## create a grouping variable of length 100
> a <- factor(sample(1:5, 100, replace = TRUE), levels = 1:5)
> a
  [1] 2 3 3 3 4 2 3 5 1 4 3 4 4 4 5 3 5 4 1 1 1 1 3 2 3 5 3 2 5 4 2 3 5 5 3 1 3
 [38] 3 5 5 1 5 4 3 3 4 4 5 1 3 5 3 2 1 1 2 1 2 2 5 4 2 2 4 3 5 1 2 2 3 4 4 2 2
 [75] 2 4 5 5 1 5 1 5 3 1 4 4 1 1 2 5 4 2 1 2 3 3 1 1 3 4
Levels: 1 2 3 4 5
> levels(a) <- LETTERS[1:5]
> a
```

```
  [1] B C C C D B C E A D C D D D E C E D A A A A C B C E C B E D B C E E C A C
 [38] C E E A E D C C D D E A C E C B A A B A B B E D B B D C E A B B C D D B B
 [75] B D E E A E A E C A D D A A B E D B A B C C A A C D
Levels: A B C D E
> plot(a)
```



Formula methods. An important theme in R is the use of a *formula language* for expressing various relationships. It is used most commonly for statistical modeling, but is also widely used for graphics. In the following example, the variable on the horizontal axis is categorical, so the resulting plot is a box-and-whisker plot. If both variables had been numeric, the result would have been a scatterplot.

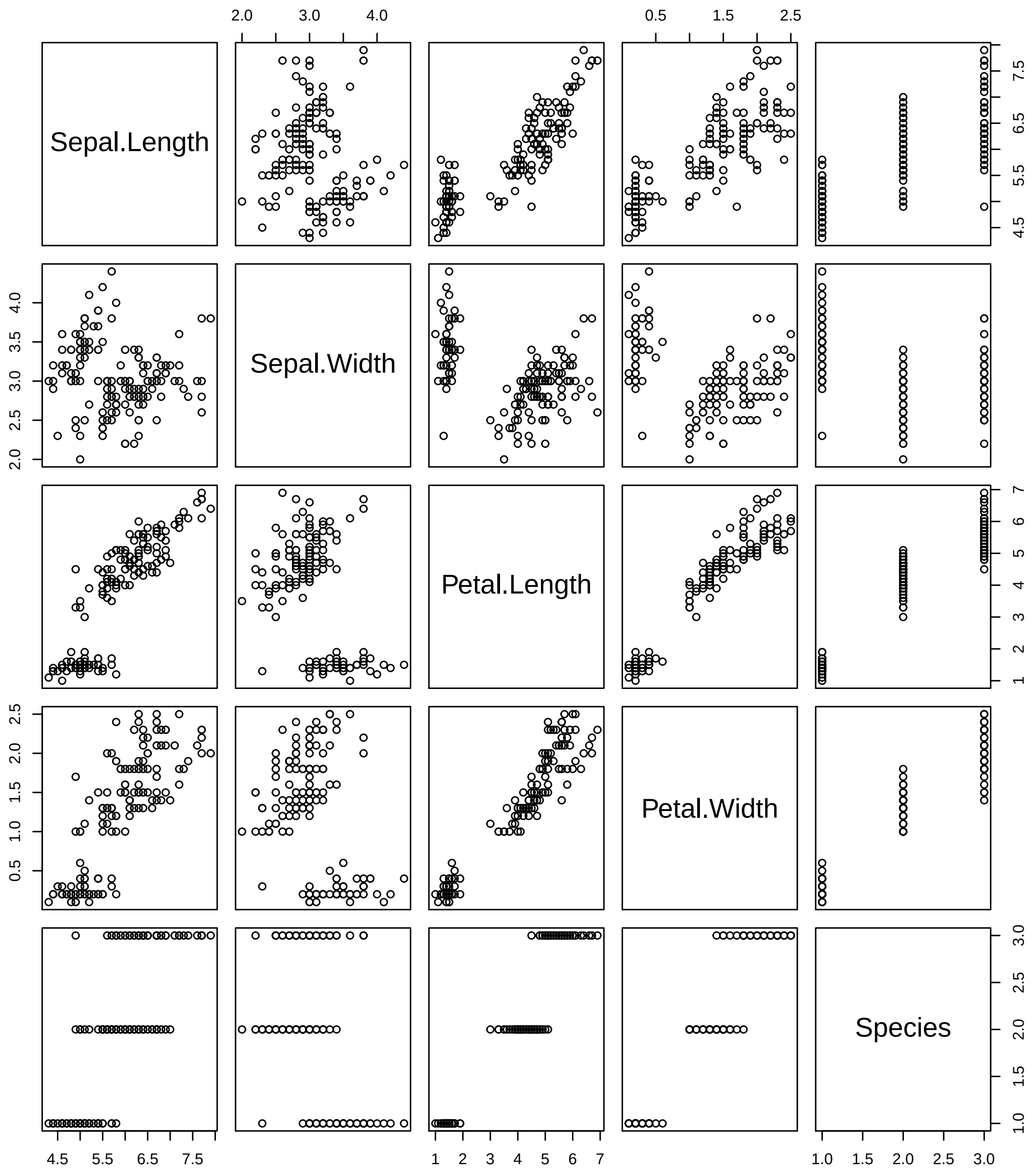
```
> plot(y ~ a)
```



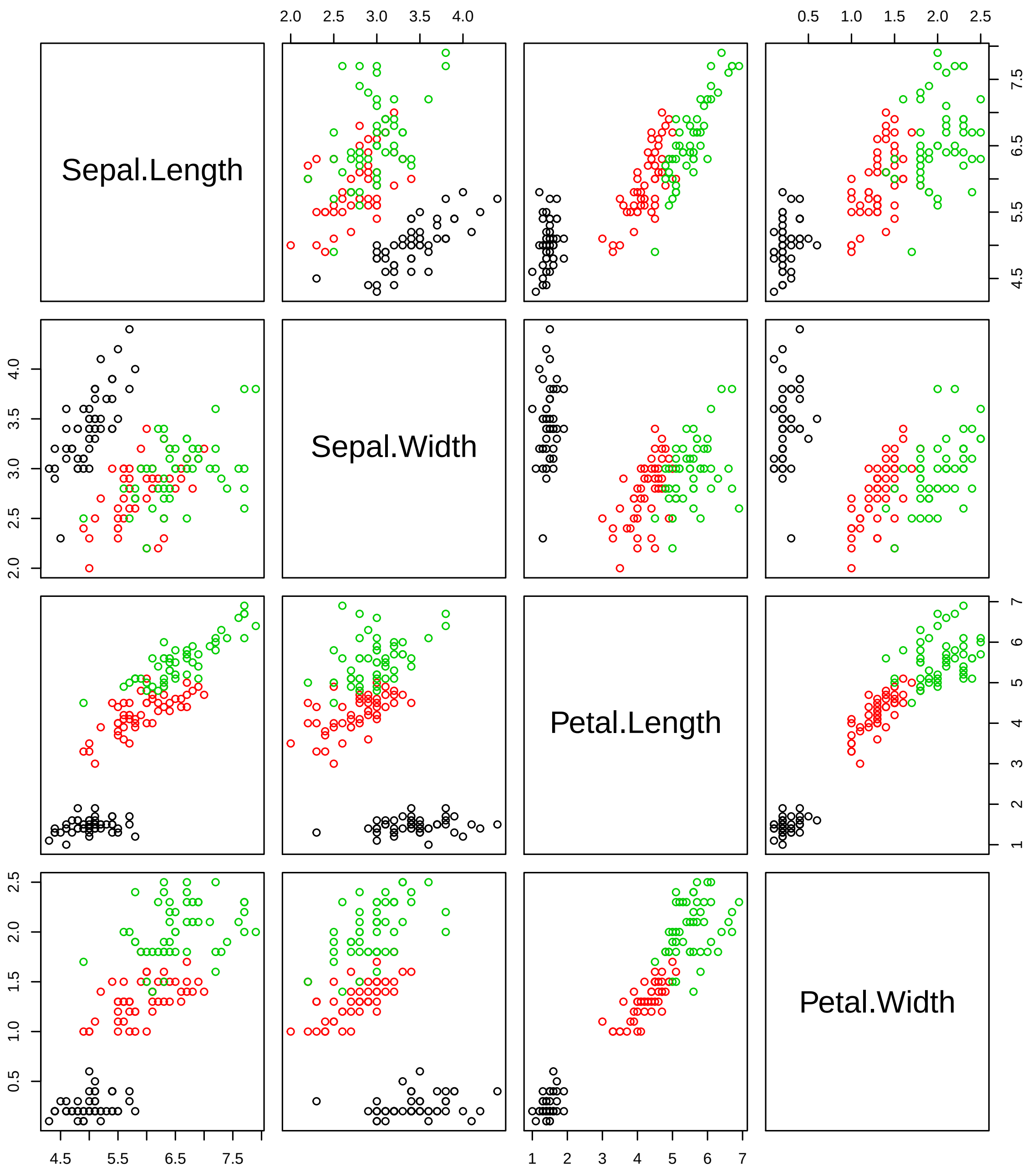
See `?formula` for a formal description of formula objects; we will see more of them later.

Data frames. The `plot()` method for "data.frame" objects produces a *scatterplot matrix*, a matrix of all pairwise bivariate scatterplots.

```
> plot(iris)
```

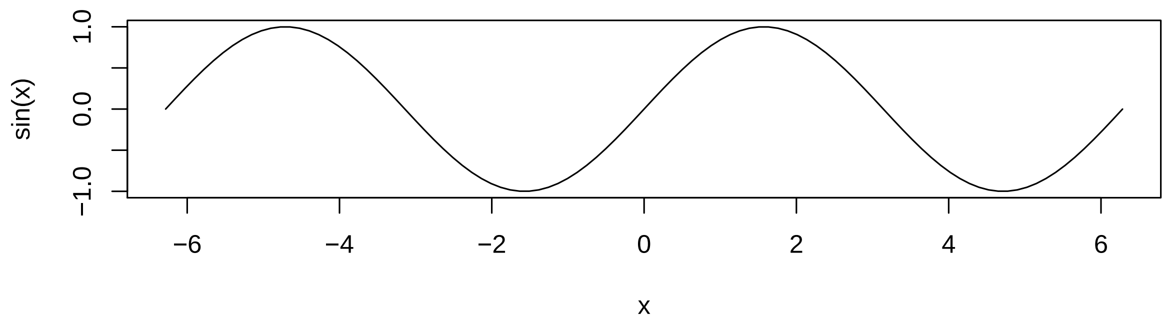


```
> plot(iris[1:4], col = as.numeric(iris$Species)) # color by Species
```

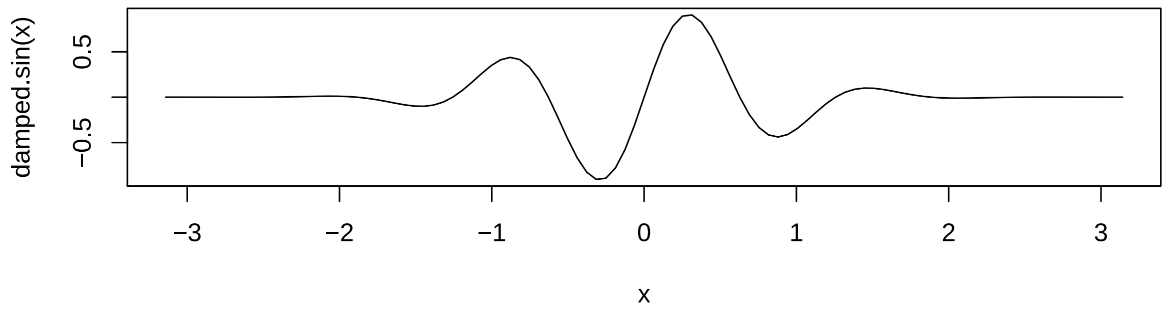


Functions. Functions of one variable can be plotted using the `plot.function()` method.

```
> plot(sin, from = -2 * pi, to = 2 * pi)
```



```
> damped.sin <- function(x) sin(5 * x) * exp(-x^2) ## New function
> plot(damped.sin, from = -pi, to = pi)
```



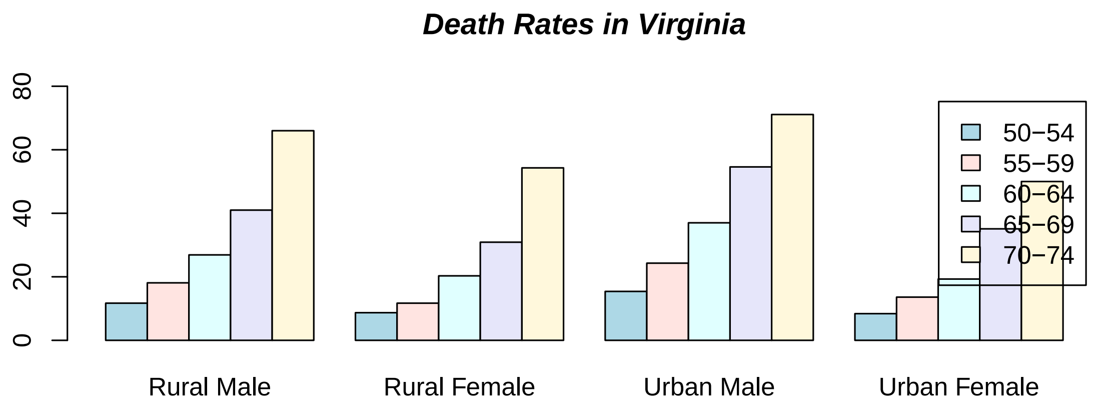
Other common high-level graphics functions. So far, we have only discussed the catch-all generic `plot()` function, but there are many high-level functions for specific types of displays. We have already seen the output produced by some of them, as they are used by the `plot()` generic when appropriate.

Exercise 2. *Use the following functions to reproduce the corresponding plots shown above. Feel free to look at documentation and examples if necessary.*

- *`pairs()`: scatterplot matrix*
- *`barplot()`: bar plots*
- *`boxplot()`': box-and-whisker plots*
- *`curve()`: function plots*

Bar plots. `barplot()` can be used to explicitly produce bar plots.

```
> data(VADeaths)
> barplot(VADeaths, beside = TRUE,
          col = c("lightblue", "mistyrose", "lightcyan", "lavender", "cornsilk"),
          legend.text = rownames(VADeaths), ylim = c(0, 80))
> title(main = "Death Rates in Virginia", font.main = 4)
```



Exercise 3. *Modify the call above to improve legend positioning, so that it does not overlap the bars.*

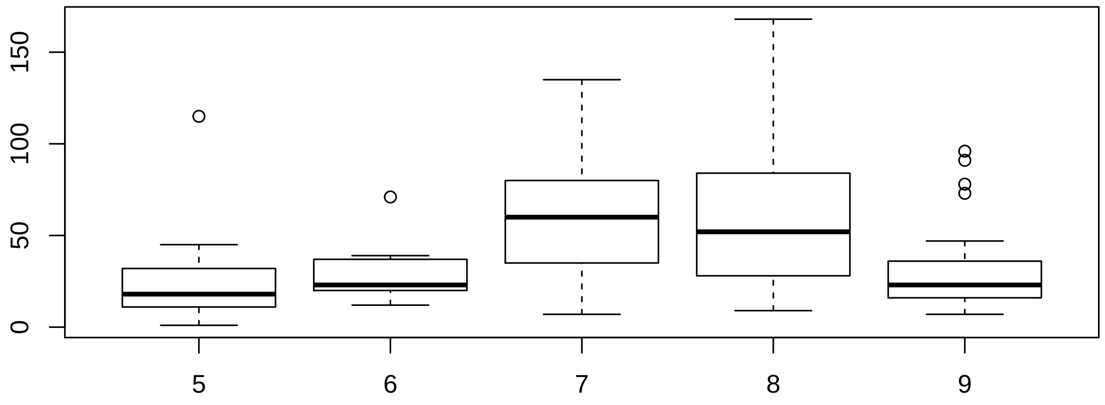
The return value of `barplot()` can be useful, and plotting can be suppressed if one is only interested in the return value.

```
> barplot(VADeaths, plot = FALSE)
[1] 0.7 1.9 3.1 4.3
> barplot(VADeaths, plot = FALSE, beside = TRUE)
     [,1] [,2] [,3] [,4]
[1,]  1.5  7.5 13.5 19.5
[2,]  2.5  8.5 14.5 20.5
[3,]  3.5  9.5 15.5 21.5
[4,]  4.5 10.5 16.5 22.5
[5,]  5.5 11.5 17.5 23.5
```

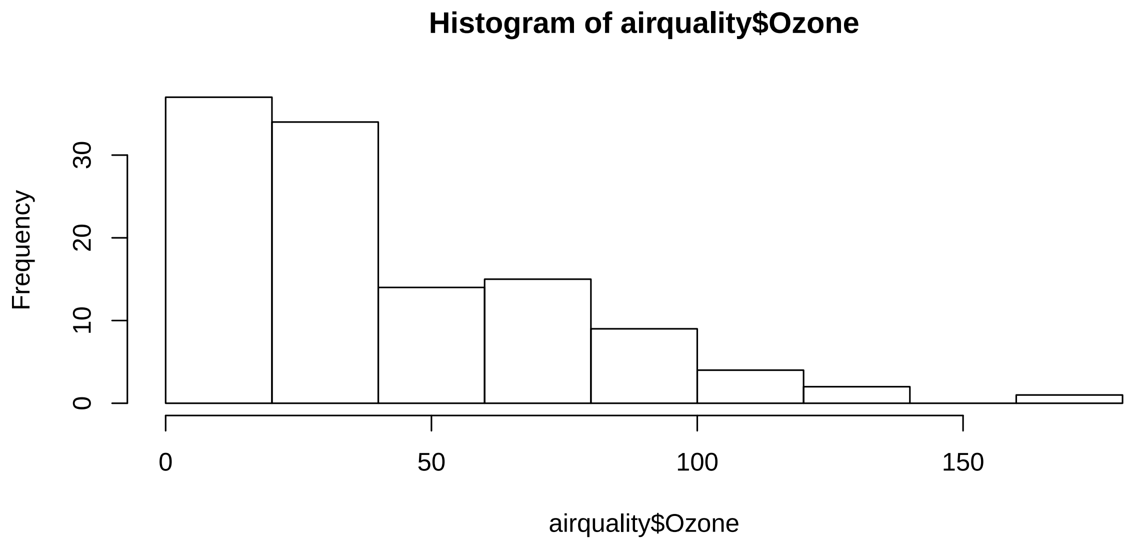
Box and whisker plots and histograms. The same is true for `boxplot()`, which produces box-and-whisker plots, and `hist()`, which produces histograms.

```
> bxp.stats <- with(airquality, boxplot(Ozone ~ factor(Month)))
> bxp.stats
```

```
$stats
     [,1] [,2] [,3] [,4] [,5]
[1,]    1   12    7    9    7
[2,]   11   20   35   28   16
[3,]   18   23   60   52   23
[4,]   32   37   80   84   36
[5,]   45   39  135  168   47
attr(,"class")
        5
"integer"

$n
[1] 26  9 26 26 29

$conf
         [,1]     [,2]     [,3]     [,4]     [,5]
[1,] 11.49287 14.04667 46.05614 34.64764 17.13203
[2,] 24.50713 31.95333 73.94386 69.35236 28.86797

$out
[1] 115  71  96  78  73  91

$group
[1] 1 2 5 5 5 5

$names
[1] "5" "6" "7" "8" "9"
```



```
> h <- hist(airquality$Ozone)
> str(h)
List of 7
 $ breaks     : num [1:10] 0 20 40 60 80 100 120 140 160 180
 $ counts     : int [1:9] 37 34 14 15 9 4 2 0 1
 $ intensities: num [1:9] 0.01595 0.01466 0.00603 0.00647 0.00388 ...
 $ density    : num [1:9] 0.01595 0.01466 0.00603 0.00647 0.00388 ...
 $ mids       : num [1:9] 10 30 50 70 90 110 130 150 170
 $ xname      : chr "airquality$Ozone"
 $ equidist   : logi TRUE
 - attr(*, "class")= chr "histogram"
```

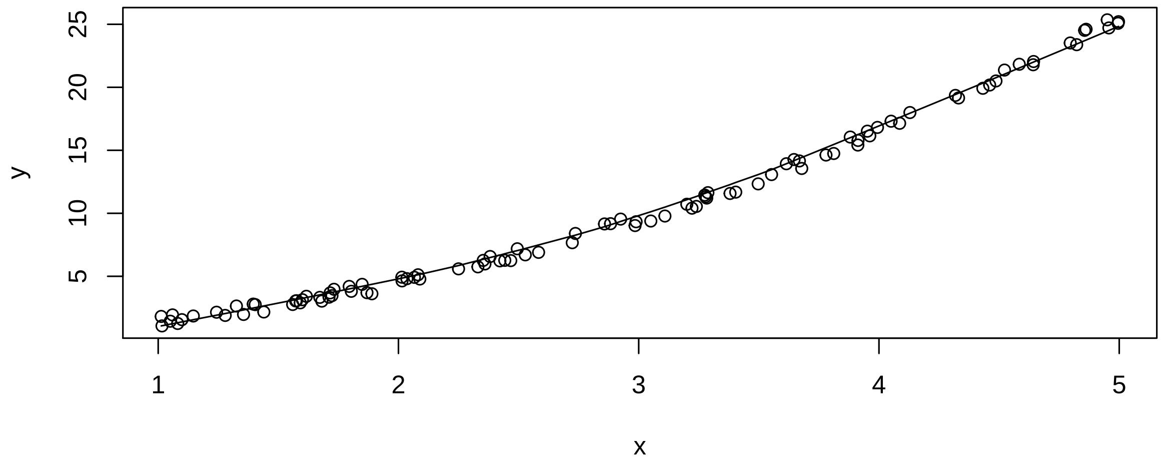


Low level functions. High-level graphics functions produce basic versions of common statistical graphs, but often we want to fine-tune them in different ways to get something more relevant to our purposes. The standard practice in traditional R graphics is to incrementally add components to an existing graph. The functions to do this are called low-level functions — distinguished by the fact that they don't create new plots themselves. In particular, they use the co-ordinate system from the existing graph. The most commonly used ones are `lines()`, `points()`, and `text()`.

A common use of this feature is to add a representation of some statistical model fit using the data. For example, the following shows a LOWESS curve (a non-parametric regression method).

```
> plot(x, y)
> lws.fit <- lowess(x, y)
> str(lws.fit)
List of 2
 $ x: num [1:100] 1.01 1.02 1.05 1.06 1.08 ...
 $ y: num [1:100] 1.06 1.08 1.2 1.24 1.32 ...
```

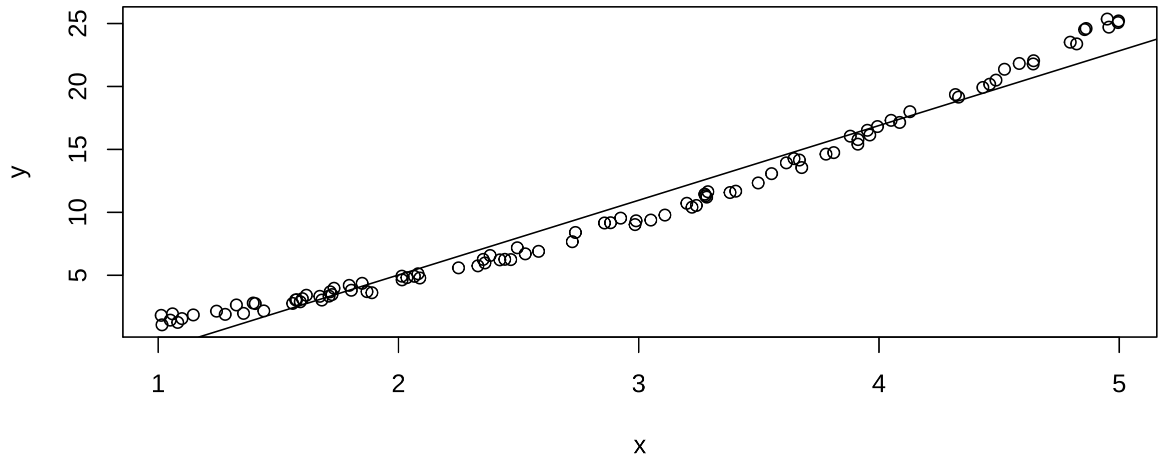
```
> lines(lws.fit, col = "black")
```



Recall that `plot()` can be used to plot a list with components `x` and `y`. The same is true for `lines()` etc. We have used the fact that `lowess()` returns a list suitable for this purpose.

A simpler and more common example is to fit and plot a linear regression to the data (which is of course not appropriate in this case).

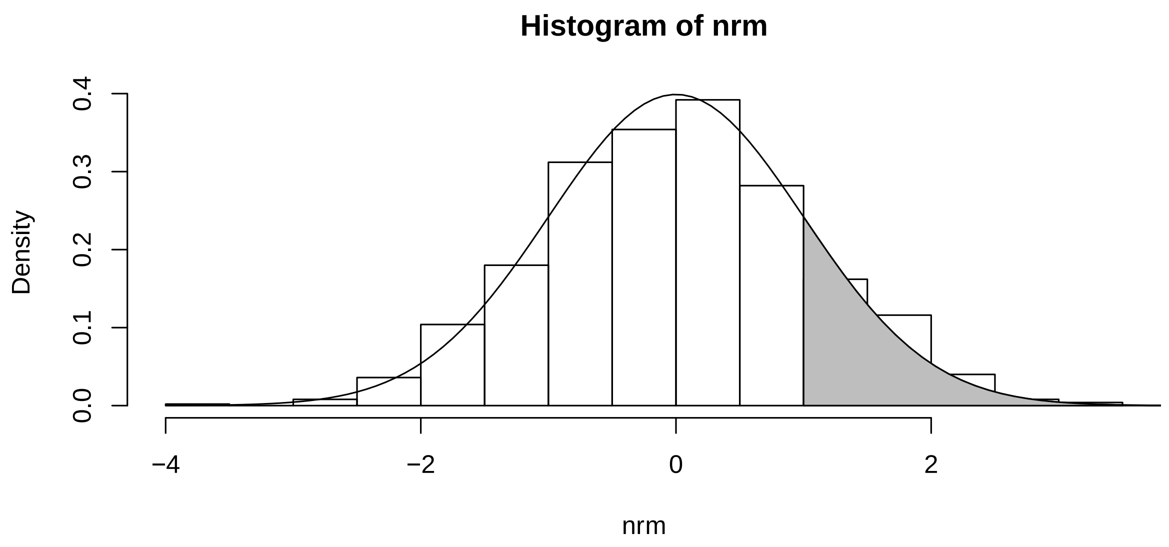
```
> plot(x, y)
> abline(lm(y ~ x))
```



We will learn about `lm()` in another tutorial. `abline()` can also draw various kinds of pre-defined (as opposed to data-dependent) straight lines, and `curve()` (already seen earlier) can draw mathematical curves.

Shading and Polygons. Arbitrary geometric shapes can be drawn and shaded using `polygon()`. For example, the following code draws the histogram from a normal random sample, adds the standard normal density curve and shades the area under the normal curve between 1 and 4.

```
> nrm <- rnorm(1000) ## simulation from standard normal
> hist(nrm, freq = FALSE)
> curve(dnorm, add = TRUE)
> ## shade area between 1 and 4
> tt <- seq(from = 1, to = 4, length = 30)
> dtt <- dnorm(tt)
> polygon(x = c(1, tt, 4), y = c(0, dtt, 0), col = "gray")
```



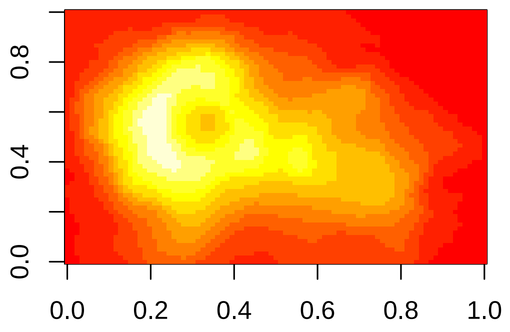
Interacting with plots. Standard R graphics has some very basic support for interaction, mainly through the functions `locator()` and `identify()`.

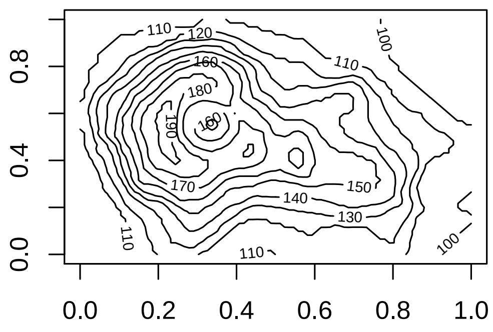
```
> plot(x, y)
> ## left-click to select points, right-click to stop
> identify(x, y)
> ## click on 5 points to make a polygon out of them
> polygon(locator(5))
```

There are interfaces to external software with better interaction facilities.

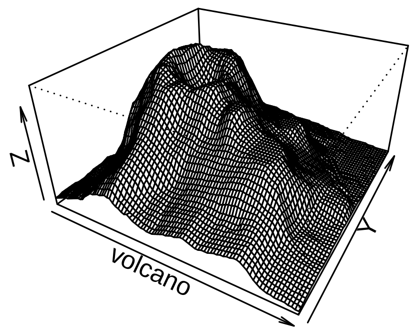
Visualizing three-dimensional (matrix) data. Traditional R graphics has some functions to visualize three-dimensional surfaces represented in the form of a matrix.

```
> str(volcano) ## heights of a volcano in New Zealand

 num [1:87, 1:61] 100 101 102 103 104 105 105 106 107 108 ...

> par(mfrow = c(1, 2)) ## multiple plots in a page
> image(volcano)
> contour(volcano)
```





```
> ## Perspective Plots
> persp(volcano, theta = 30, phi = 30, expand = 0.5)
```



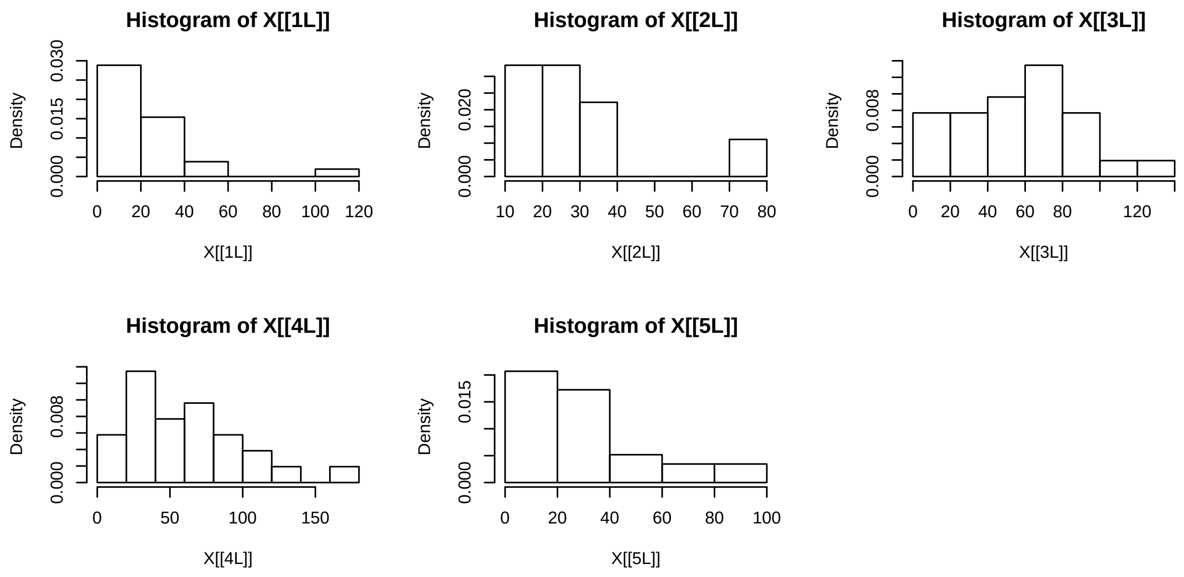
Further exploration. For further study, consult the various help pages in the graphics package; see

```
> library(help = graphics)
```

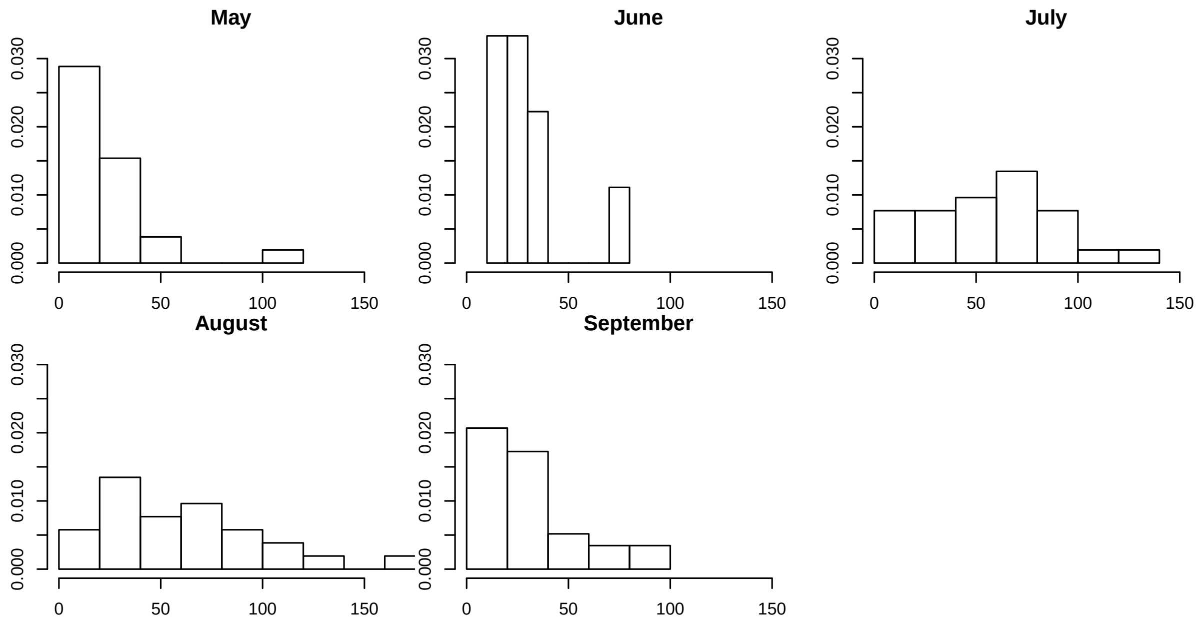
Weaknesses of traditional graphics. The traditional graphics model is considerably limited by its design. A basic assumption in the approach is that there will be a single figure area in the center with a single coordinate system, and margins will be used for axis annotation and labels. Not all graphical designs fit into this paradigm.

One example is a graphical design known as "small multiples" or "conditioned plots", where one page contains multiple plots grouped by some categorical variable. Here is a very poor attempt to implement such a design using traditional graphics.

```
> par(mfrow = c(2, 3))
> s <- with(airquality,
            split(Ozone, factor(Month, levels = 1:12, labels = month.name)))
> s <- s[sapply(s, length) > 0]
> str(s)
List of 5
 $ May      : int [1:31] 41 36 12 18 NA 28 23 19 8 NA ...
 $ June     : int [1:30] NA NA NA NA NA NA 29 NA 71 39 ...
 $ July     : int [1:31] 135 49 32 NA 64 40 77 97 97 85 ...
 $ August   : int [1:31] 39 9 16 78 35 66 122 89 110 NA ...
 $ September: int [1:30] 96 78 73 91 47 32 20 23 21 24 ...
> invisible(lapply(s, hist, freq = FALSE))
```



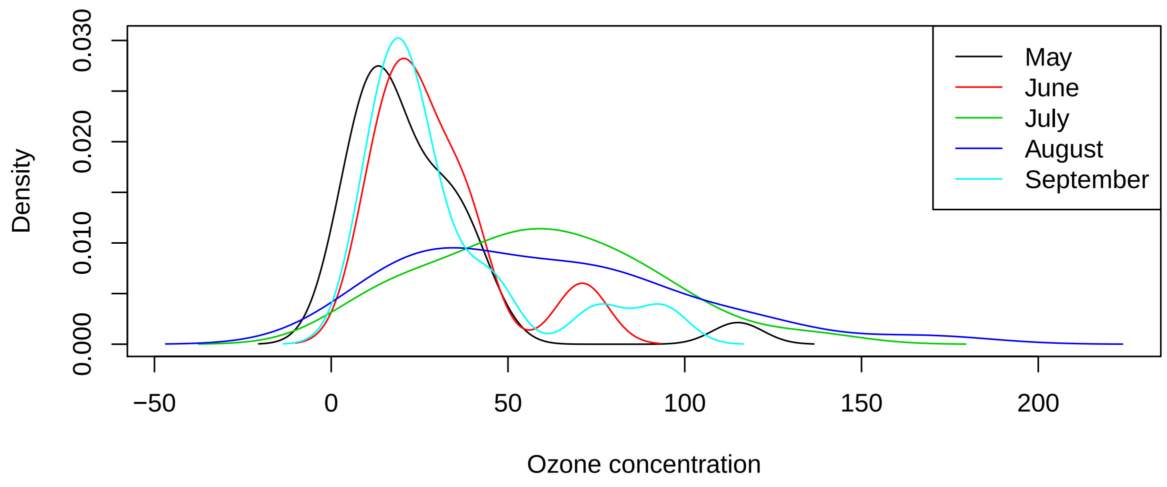
Exercise 4. *Reproduce the following plot, which is a modification of the previous plot, but with the same axis limits in each sub-plot, better labels, and more efficient use of available space. For the last requirement, you will need to use a* `par()` *setting we have not yet encountered.*



Do not worry if you cannot get everything right; the main point of this exercise is to show that implementing such designs is difficult with traditional graphics. In the next tutorial we will learn about a different high-level graphics system called lattice. Try the following code to get a preview of the kind of plots lattice can produce.

```
> library(lattice)
> airquality$fmonth <- factor(airquality$Month, levels = 1:12, labels = month.name)
> histogram(~Ozone | fmonth, data = airquality)
```

Exercise 5. *Histograms are "density estimators", in the sense that they estimate the unknown probability density of the variable being plotted. A more sophisticated and computer-intensive method for the same purpose is Kernel density estimation, which can be performed using the* `density()` *function; for example, try* `plot(density(airquality£Ozone, na.rm = TRUE))`*. Your exercise is to plot superposed density estimates of* `Ozone` *by month, to produce a plot similar to the following. Your first step could be to compute the per-group densities using* `dlist <- lapply(s, density, na.rm = TRUE)`



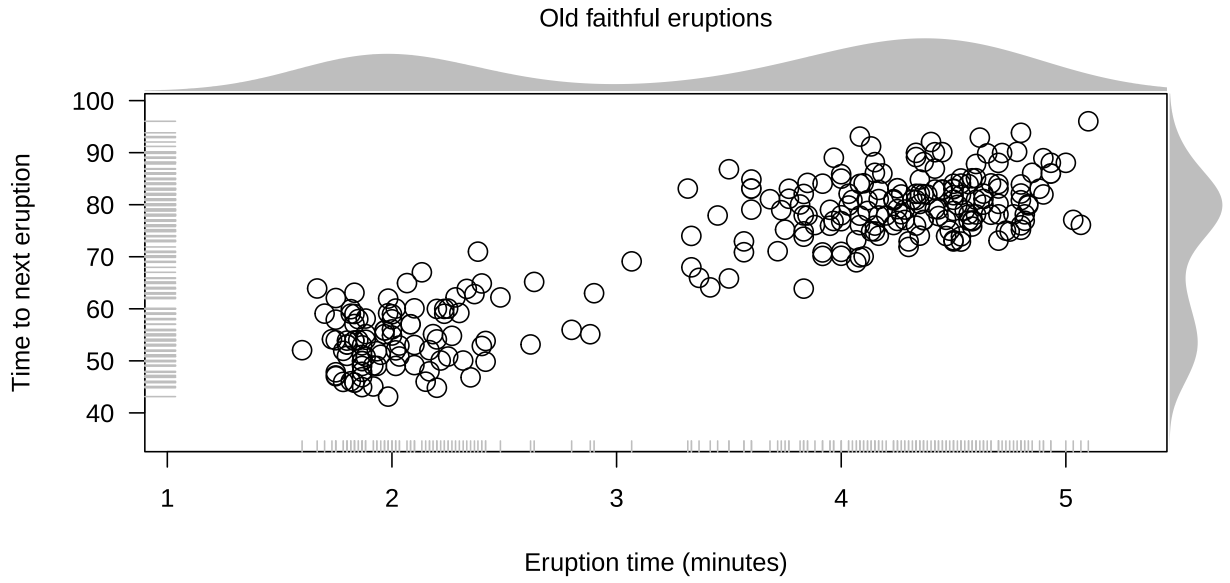
Grid graphics

Grid graphics is a more flexible low-level graphics toolbox. It does not provide high-level functions itself, but other packages use it to provide them; most notably the lattice and ggplot2 packages. The only real drawback of grid is that it is slower than traditional graphics, and not all traditional plots have a replacement using Grid graphics.

We will not go into the details of grid, as it is a considerable topic. Instead, we will just give a glimpse of its flexibility using a practical example. Our goal is to implement an *enhanced scatter plot* with the following features:

- Regular scatter plot showing bivariate relation
- "Rug" on left/bottom showing marginal scatter
- Density estimate on right/top

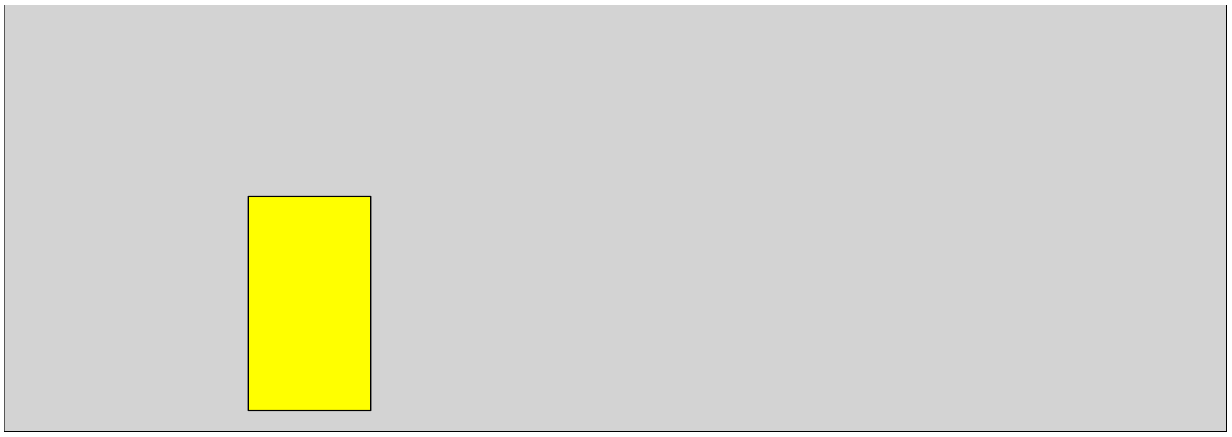
This is a simple design, but difficult with standard graphics.



Viewports. Viewports are a central concept in Grid graphics. They are essentially rectangular subregions of the plotting area. They can be nested within other viewports, forming a tree of viewports. The initial blank graphics area is the ROOT viewport. New viewports can be defined relative to parent either by positioning, or in terms of a layout.

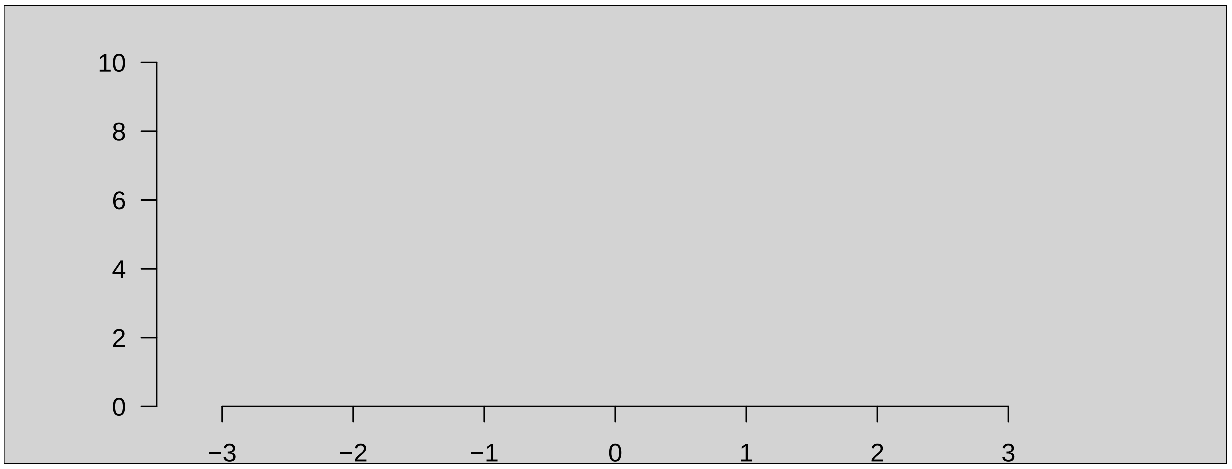
Viewports are created by the `viewport()` function, and made active by `pushViewport()`. `upViewport()` goes up along the viewport tree.

```
> grid.rect(gp = gpar(fill = 'lightgrey'))
> pushViewport(viewport(x = 0.25, y = 0.3, width = 0.1, height = 0.5))
> grid.rect(gp = gpar(fill = 'yellow'))
```



When creating a viewport, we can specify a *native coordinate system* for it.

```
> grid.rect(gp = gpar(fill = 'lightgrey'))
> pushViewport(viewport(x = 0.5, y = 0.5,
                        width = 0.75, height = 0.75,
                        xscale = c(-3.5, 3.5),
                        yscale = c(0, 10)))
> grid.xaxis()
> grid.yaxis()
```



Units. Another fundamental concept is that of units of length. Grid can specify lengths in various ways.

`"native"`	Native coordinates
`"npc"`	Normalized Parent Coordinates
`"snpc"`	'Square' NPC
`"inches"`, `"cm"`, `"points"`, etc.	absolute lengths
`"char"`	Character size (depends on font details)
`"lines"`	Height of a lines of text
`"strwidth"`, `"strheight"`	Width or height of a string
`"grobwidth"`, `"grobheight"`	Width or height of a 'grid object' (details later)

Primitives. Low-level functions in traditional graphics have analogs in grid. There are two versions of each function, one that produces output, and one that produces an object without actually plotting it. These objects can be queried to determine their height and width, so that appropriate space can be allocated for them.

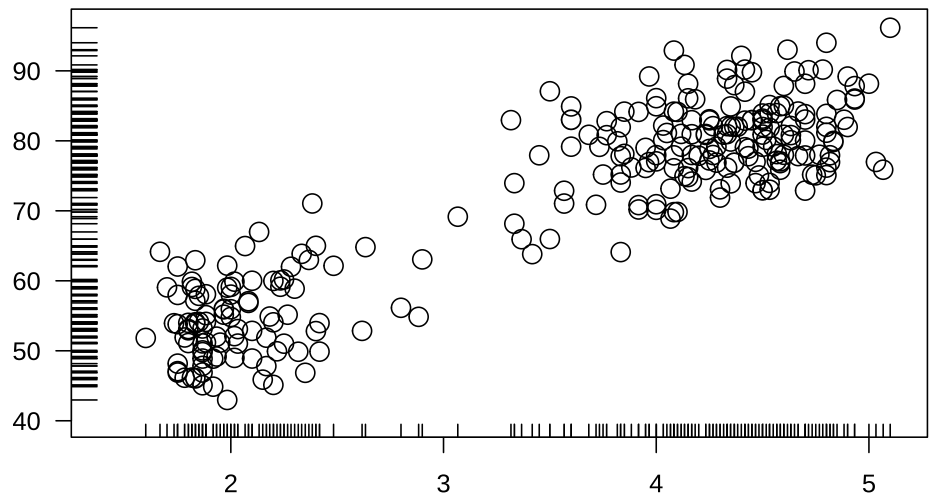
Type	Function producing output	Function producing object
Points	`grid.points`	`pointsGrob`
Lines	`grid.lines`	`linesGrob`
Text	`grid.text`	`textGrob`
Rectangles	`grid.rect`	`rectGrob`
Circles	`grid.circle`	`circleGrob`
Polygons	`grid.polygon`	`polygonGrob`
Segments	`grid.segments`	`segmentsGrob`
X-axis	`grid.xaxis`	`xaxisGrob`
Y-axis	`grid.yaxis`	`yaxisGrob`
Arrows	`grid.arrows`	`arrowsGrob`

First attempt. The data we have plotted is from the faithful dataset, which contains eruption times and inter-eruption intervals lengths for the famous Old Faithful geyser in the USA. We first perform some relevant computations.

```
> str(faithful)

'data.frame':        272 obs. of  2 variables:
 $ eruptions: num  3.6 1.8 3.33 2.28 4.53 ...
 $ waiting  : num  79 54 74 62 85 55 88 85 51 85 ...

> x <- faithful$eruptions
> y <- jitter(faithful$waiting)
> xrng <- range(x)
> yrng <- range(y)
> xlim <- xrng + 0.05 * c(-2, 1) * diff(xrng)
> ylim <- yrng + 0.05 * c(-2, 1) * diff(yrng)
> xdens <- density(x)
> ydens <- density(y)
> xdens.max <- max(xdens$y)
> ydens.max <- max(ydens$y)
>
```

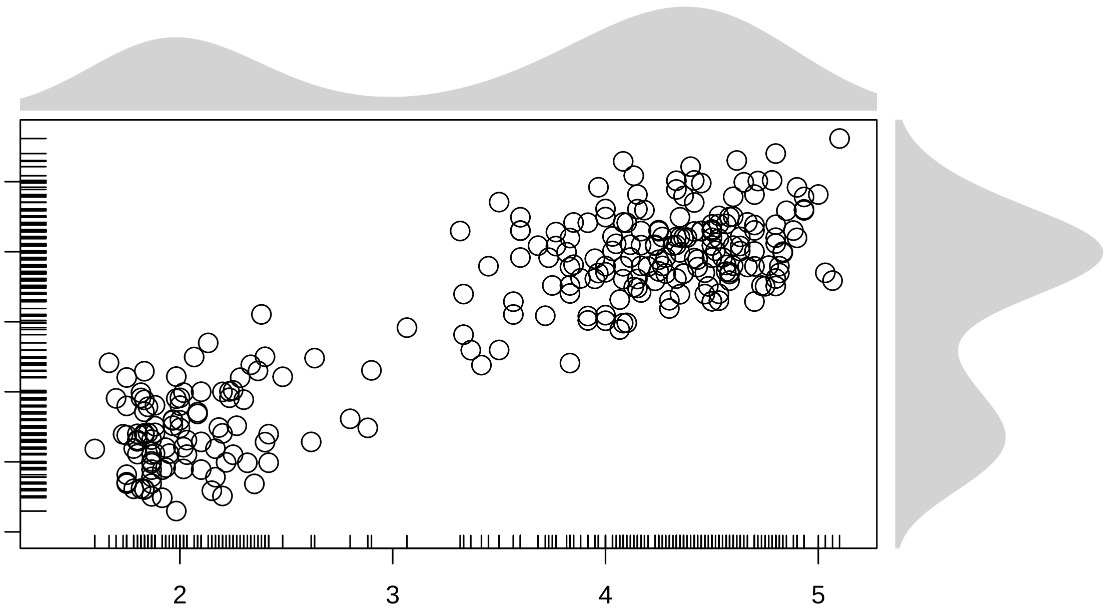
We can now use these to make a first attempt at our desired plot.

```
> pushViewport(viewport(x = 0.45, y = 0.45, width = 0.7, height = 0.7,
                        xscale = xlim, yscale = ylim))
> grid.points(x, y, default.units = "native")
> grid.rect()
> grid.xaxis()
> grid.yaxis()
> grid.segments(x0 = unit(x, "native"), y0 = unit(0, "npc"),
                x1 = unit(x, "native"), y1 = unit(0.03, "npc"))
> grid.segments(x0 = unit(0, "npc"), y0 = unit(y, "native"),
                x1 = unit(0.03, "npc"), y1 = unit(y, "native"))
```



To this, we can add the densities by creating two new viewports and plotting the densities inside.

```
> upViewport(1)
> pushViewport(viewport(x = 0.45, y = 0.90, width = 0.7, height = 0.17,
                        xscale = xlim, yscale = c(0, xdens.max),
                        clip = "on"))
> grid.polygon(x = xdens$x, y = xdens$y, default.units = "native",
               gp = gpar(fill = "lightgrey", col = "transparent"))
> upViewport(1)
> pushViewport(viewport(x = 0.90, y = 0.45, width = 0.17, height = 0.7,
                        xscale = c(0, ydens.max), yscale = ylim,
                        clip = "on"))
> grid.polygon(x = ydens$y, y = ydens$x, default.units = "native",
               gp = gpar(fill = "lightgrey", col = "transparent"))
> upViewport(1)
```



There are several problems with this approach. It is not easily generalizable to other datasets. We would also prefer to leave space for labels only if they are present. The way around the first problem is to write a more general function. For the second problem, we must use an alternate method of specifying viewports using *layouts*, where a parent viewport is divided into rows and columns with more flexible specification of heights and widths.

A detailed discussion of this approach would take too long. If you are interested, look at the file `"esplot.R"`. Grid itself has many more features, see package documentation for details.