R-Basic Graphs

R-Basic Graphs

- R offers a remarkable variety of graphics. To get an idea, one can type demo(graphics) or demo(persp).
- It is not possible to detail here the possibilities of R in terms of graphics, particularly since each graphical function has a large number of options making the production of graphics very flexible.
- The way graphical functions work deviates substantially.
- The result of a graphical function cannot be assigned to an object but is sent to a graphical device.

R-Basic Graphs

- A graphical device is a graphical window or a file.
- There are two kinds of graphical functions:
  - The high-level plotting functions which create a new graph
  - The low-level plotting functions which add elements to an existing graph
- The graphs are produced with respect to graphical parameters which are defined by default and can be modified with the function par.

Managing graphics

Opening several graphical devices

- When a graphical function is executed, if no graphical device is open, R opens a graphical window and displays the graph.
- A graphical device may be open with an appropriate function. The list of available graphical devices depends on the operating system.
- The graphical windows are called X11 under Unix/Linux and windows under Windows.
- In all cases, one can open a graphical window with the command x11() which also works under Windows because of an alias towards the command windows().

Managing graphics

- A graphical device which is a file will be open with a function depending on the format: postscript(), pdf(), png(),
- The list of available graphical devices can be found with ?device.
- The last open device becomes the active graphical device on which all subsequent graphs are displayed.
- The function dev.list() displays the list of open devices:

```
> x11(); x11(); pdf()
> dev.list()
```

Managing graphics

- The figures displayed are the device numbers which must be used to change the active device.
- To know what is the active device:

```
> dev.cur()
```

- and to change the active device:

```
> dev.set(3)
```

- The function dev.off() closes a device: by default the active device is closed, otherwise this is the one which number is given as argument to the function.
- R then displays the number of the new active device:

```
> dev.off(2)
> dev.off()
```

Managing graphics

- A Windows Meta file device can be open with the function win.metafile, and a menu "History" displayed when the graphical window is selected allowing recording of all graphs drawn during a session (by default, the recording system is off, the user switches it on by clicking on "Recording" in this menu).

Partitioning a graphic

- The function split.screen partitions the active graphical device.

```
> split.screen(c(1, 2))
```

- divides the device into two parts which can be selected with screen(1) or screen(2); erase.screen() deletes the last drawn graph.
- A part of the device can itself be divided with split.screen() leading to the possibility to make complex arrangements.
- These functions are incompatible with others (such as layout or coplot) and must not be used with multiple graphical devices.
- Their use should be limited to graphical exploration of data.

Partitioning a graphic

- The function layout partitions the active graphic window in several parts where the graphs will be displayed successively.
- Its main argument is a matrix with integer numbers indicating the numbers of the "sub-windows".
- For example, to divide the device into four equal parts:

```
> layout(matrix(1:4, 2, 2))
```

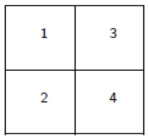
- It is possible to create this matrix previously allowing to better visualize how the device is divided:

```
> mat <- matrix(1:4, 2, 2)
> mat
        [,1]   [,2]
[1,]    1      3
[2,]    2      4
> layout(mat)
```

Partitioning a graphic

- To actually visualize the partition created, one can use the function layout.show with the number of sub-windows as argument (here 4).
- In this example, we will have:

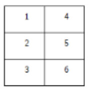
```
> layout.show(4)
```

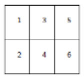


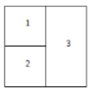
Partitioning a graphic

- The following examples show some of the possibilities offered by layout().

```
> layout(matrix(1:6, 3, 2))
> layout.show(6)

> layout(matrix(1:6, 2, 3))
> layout.show(6)

> m <- matrix(c(1:3, 3), 2, 2)
> layout(m)
> layout.show(3)
```





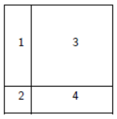


Partitioning a graphic

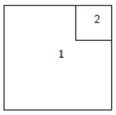
- In all these examples, we have not used the option byrow of matrix(), the sub-windows are thus numbered column-wise; one can just specify matrix(..., byrow=TRUE) so that the sub-windows are numbered row-wise.
- The numbers in the matrix may also be given in any order, for example, matrix(c(2, 1, 4, 3), 2, 2).
- By default, layout() partitions the device with regular heights and widths. This can be modified with the options widths and heights.
- These dimensions are given relatively.

Partitioning a graphic

```
> m <- matrix(1:4, 2, 2)
> layout(m, widths=c(1, 3),
                        heights=c(3, 1))
> layout.show(4)
```



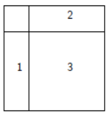
```
> m <- matrix(c(1,1,2,1),2,2)
> layout(m, widths=c(2, 1),
                        heights=c(1, 2))
> layout.show(2)
```



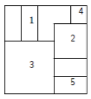
Partitioning a graphic

- Finally, the numbers in the matrix can include zeros giving the possibility to make complex (or even esoterical) partitions.

```
> m <- matrix(0:3, 2, 2)
> layout(m, c(1, 3), c(1, 3))
> layout.show(3)
```



```
> m <- matrix(scan(), 5, 5)
1: 0 0 3 3 3 1 1 3 3 3
11: 0 0 3 3 3 0 2 2 0 5
21: 4 2 2 0 5
26:
Read 25 items
> layout(m)
> layout.show(5)
```



High Level Graphical functions

- An overview of the high-level graphical functions in R.

Function	Description
plot(x)	plot of the values of x (on the y-axis) ordered on the x-axis
plot(x, y)	bivariate plot of x (on the x-axis) and y (on the y-axis)
sunflowerplot(x, y)	id. but the points with similar coordinates are drawn as a flower which petal number represents the number of points
pie(x)	circular pie-chart
boxplot(x)	"box-and-whiskers" plot
stripchart(x)	plot of the values of x on a line (an alternative to boxplot() for small sample sizes)
coplot(x~y | z)	bivariate plot of x and y for each value (or interval of values) of z

High Level Graphical functions

Function	Description
interaction.plot (f1, f2, y)	if f1 and f2 are factors, plots the means of y (on the y-axis) with respect to the values of f1 (on the x-axis) and of f2 (different curves); the option fun allows to choose the summary statistic of y (by default fun=mean)
matplot(x,y)	bivariate plot of the rst column of x vs. the first one of y, the second one of x vs. the second one of y, etc.
dotchart(x)	if x is a data frame, plots a Cleveland dot plot (stacked plots line-by-line and column-by-column)
fourfoldplot(x)	visualizes, with quarters of circles, the association between two dichotomous variables for different populations (x must be an array with dim=c(2, 2, k), or a matrix with dim=c(2, 2) if k = 1)

High Level Graphical functions

Function	Description
assocplot(x)	Cohen-Friendly graph showing the deviations from independence of rows and columns in a two dimensional contingency table
mosaicplot(x)	`mosaic' graph of the residuals from a log-linear regression of a contingency table
pairs(x)	if x is a matrix or a data frame, draws all possible bivariate plots between the columns of x
plot.ts(x)	if x is an object of class "ts", plot of x with respect to time, x may be multivariate but the series must have the same frequency and dates
ts.plot(x)	id. but if x is multivariate the series may have dierent dates and must have the same frequency
hist(x)	histogram of the frequencies of x

High Level Graphical functions

Function	Description
barplot(x)	histogram of the values of x
qqnorm(x)	quantiles of x with respect to the values expected under a normal Law
qqplot(x, y)	quantiles of y with respect to the quantiles of x
contour(x, y, z)	contour plot (data are interpolated to draw the curves), x and y must be vectors and z must be a matrix so that dim(z)=c(length(x), length(y)) (x and y may be omitted)
filled.contour (x, y, z)	id. but the areas between the contours are coloured, and a legend of the colours is drawn as well
image(x, y, z)	id. but the actual data are represented with colours

High Level Graphical functions

Function	Description
persp(x, y, z)	id. but in perspective
stars(x)	if x is a matrix or a data frame, draws a graph with segments or a star where each row of x is represented by a star and the columns are the lengths of the segments
symbols(x, y, ...)	draws, at the coordinates given by x and y, symbols (circles, squares, rectangles, stars, thermometers or “boxplots”) which sizes, colours, etc, are specified by supplementary arguments
termplot(mod.obj)	plot of the (partial) effects of a regression model (mod.obj)

Common Graphical Parameters

- For each function, the options may be found with the on-line help in R.
- Some of these options are identical for several graphical functions; here are the main ones (with their possible default values):

Function	Description
add=FALSE	if TRUE superposes the plot on the previous one (if it exists)
axes=TRUE	if FALSE does not draw the axes and the box
type="p"	specifies the type of plot, "p": points, "l": lines, "b": points connected by lines, "o": id. but the lines are over the points, "h": vertical lines, "s": steps, the data are represented by the top of the vertical lines, "S": id. But the data are represented by the bottom of the vertical Lines
xlim=, ylim=	specifies the lower and upper limits of the axes, for example with xlim=c(1, 10) or xlim=range(x)
xlab=, ylab=	annotates the axes, must be variables of mode character
main=	main title, must be a variable of mode character
sub=	sub-title (written in a smaller font)

Low Level Graphical functions

- R has a set of graphical functions which affect an already existing graph: they are called low-level plotting commands.
- Here are the main ones:

Function	Description
points(x, y)	adds points (the option type= can be used)
lines(x, y)	Identical to points, but with lines
text(x, y, labels,...)	adds text given by labels at coordinates (x,y); a typical use is: plot(x, y, type="n"); text(x, y, names)
mtext(text, side=3, line=0,...)	adds text given by text in the margin specied by side (see axis() below); line species the line from the plotting area
segments(x0, y0, x1, y1)	draws lines from points (x0,y0) to points (x1,y1)

Low Level Graphical functions

Function	Description
arrows(x0, y0, x1, y1, angle= 30, code=2)	id. with arrows at points (x0,y0) if code=2, at points (x1,y1) if code=1, or both if code=3; angle controls the angle from the shaft of the arrow to the edge of the arrow head
abline(a, b)	draws a line of slope b and intercept a
abline(h=y)	draws a horizontal line at ordinate y
abline(v=x)	draws a vertical line at abcissa x
abline(lm.obj)	draws the regression line given by lm.obj
rect(x1, y1, x2, y2)	draws a rectangle which left, right, bottom, and top limits are x1, x2, y1, and y2, respectively

Low Level Graphical functions

Function	Description
polygon(x, y)	draws a polygon linking the points with coordinates given by x and y
legend(x, y, legend)	adds the legend at the point (x,y) with the symbols given by legend
title()	adds a title and optionally a sub-title
axis(side, vect)	adds an axis at the bottom (side=1), on the left (2), at the top (3), or on the right (4); vect (optional) gives the abcissa (or ordinates) where tick-marks are drawn
box()	adds a box around the current plot
rug(x)	draws the data x on the x-axis as small vertical lines
locator(n, type="n", ...)	returns the coordinates (x; y) after the user has clicked n times on the plot with the mouse; also draws symbols (type="p") or lines (type="l") with respect to optional graphic parameters (...); by default nothing is drawn (type="n")

Low Level Graphical functions

- Note the possibility to add mathematical expressions on a plot with text(x, y, expression(...)), where the function expression transforms its argument in a mathematical equation.
- For example,

```
> text(x, y, expression(p == over(1, 1+e^ (beta*x+alpha))))
```

- will display, on the plot, the following equation at the point of coordinates (x; y):

$$p = \frac{1}{1 + e^{-(\beta x+\alpha)}}$$

Low Level Graphical functions

- To include in an expression a variable we can use the functions substitute and as.expression; for example to include a value of R2 (previously computed and stored in an object named Rsquared):

```
> text(x, y, as.expression(substitute(R^2==r,
                               list(r=Rsquared))))
```

- will display on the plot at the point of coordinates (x; y):

$R^2 = 0.9856298$

Low Level Graphical functions

- To display only three decimals, we can modify the code as follows:

```
> text(x, y, as.expression(substitute(R^2==r,
                  list(r=round(Rsquared, 3)))))
```

$R^2 = 0.986$

- Finally, to write the R in italics:

```
> text(x, y, as.expression(substitute(italic(R)^2==r,
                  list(r=round(Rsquared, 3)))))
```

$\mathit{R}^2 = \mathit{0.986}$

par() function

- The presentation of graphics can be improved with graphical parameters.
- They can be used either as options of graphic functions (but it does not work for all), or with the function par to change permanently the graphical parameters, i.e. the subsequent plots will be drawn with respect to the parameters specified by the user.
- For instance, the following command:

  > par(bg="yellow")

- will result in all subsequent plots drawn with a yellow background.
- There are 73 graphical parameters, some of them have very similar functions. The exhaustive list of these parameters can be read with ?par.

par() function

Parameter	Description
adj	controls text justification with respect to the left border of the text so that 0 is left-justified, 0.5 is centred, 1 is right-justified, values > 1 move the text further to the left, and negative values further to the right; if two values are given (e.g., c(0, 0)) the second one controls vertical justify cation with respect to the text baseline
bg	specifies the colour of the background (e.g., bg="red", bg="blue"; the list of the 657 available colours is displayed with colors())
bty	controls the type of box drawn around the plot, allowed values are: "o", "l", "7", "c", "u" ou "]" (the box looks like the corresponding character); if bty="n" the box is not drawn
cex	a value controlling the size of texts and symbols with respect to the default; the following parameters have the same control for numbers on the axes, cex.axis, the axis labels, cex.lab, the title, cex.main, and the sub-title, cex.sub
col	controls the colour of symbols; as for cex there are: col.axis, col.lab, col.main, col.sub

par() function

Parameter	Description
font	an integer which controls the style of text (1: normal, 2: italics, 3: bold, 4: bold italics); as for cex there are: font.axis, font.lab, font.main, font.sub
las	an integer which controls the orientation of the axis labels (0: parallel to the axes, 1: horizontal, 2: perpendicular to the axes, 3: vertical)
lty	controls the type of lines, can be an integer (1: solid, 2: dashed, 3: dotted, 4: dotdash, 5: longdash, 6: twodash), or a string of up to eight characters (between "0" and "9") which species alternatively the length, in points or pixels, of the drawn elements and the blanks, for example lty="44" will have the same effect than lty=2
lwd	a numeric which controls the width of lines
mar	a vector of 4 numeric values which control the space between the axes and the border of the graph of the form c(bottom, left, top, right), the default values are c(5.1, 4.1, 4.1, 2.1)
mfcol	a vector of the form c(nr,nc) which partitions the graphic window as a matrix of nr lines and nc columns, the plots are then drawn in columns

par() function

Parameter	Description
mfrow	id. but the plots are then drawn in line
pch	controls the type of symbol, either an integer between 1 and 25, or any single character within ""
ps	an integer which controls the size in points of texts and symbols
pty	a character which species the type of the plotting region, "s": square, "m": Maximal
tck	a value which specifies the length of tick-marks on the axes as a fraction of the smallest of the width or height of the plot; if tck=1 a grid is drawn
tcl	id. but as a fraction of the height of a line of text (by default tcl=-0.5)
xaxt	if xaxt="n" the x-axis is set but not drawn (useful in conjunction with axis(side=1, ...))
yaxt	if yaxt="n" the y-axis is set but not drawn (useful in conjunction with axis(side=2, ...))

par() function

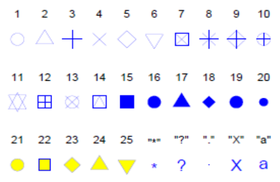


Figure 2: The plotting symbols in R (pch=1:25). The colours were obtained with the options col="blue", bg="yellow", the second option has an effect only for the symbols 21–25. Any character can be used (pch="*", "?", ".", …).

A practical example

- In order to illustrate R's graphical functionalities, let us consider a simple example of a bivariate graph of 10 pairs of random variates.
- These values were generated with:
  - > x <- rnorm(10)
  - > y <- rnorm(10)
- The wanted graph will be obtained with plot(); one will type the command:
  - > plot(x, y)

Bar Plot

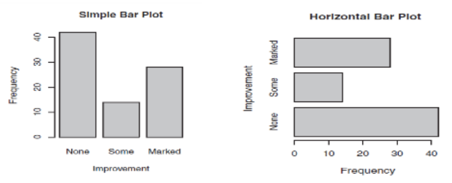
- A bar plot displays the distribution (frequency) of a categorical variable through vertical or horizontal bars.
- In its simplest form, the format of the barplot() function is

  barplot(height)
  - where height is a vector or matrix.
- The data are contained in the Arthritis data frame distributed with the vcd package.

```
> library(vcd)
> counts <- table(Arthritis$Improved)
> counts
None      Some   Marked
42        14     28
```

Simple bar plots:

```
> barplot(counts, main="Simple Bar Plot", xlab="Improvement",
                                ylab="Frequency")

> barplot(counts, main="Horizontal Bar Plot", xlab="Frequency",
                                ylab="Improvement",horiz=TRUE)
```



Bar plots with factor variables

- If the categorical variable to be plotted is a factor or ordered factor, you can create a vertical bar plot quickly with the plot() function. Because Arthritis$Improved is a factor, the code

```
> plot(Arthritis$Improved, main="Simple Bar Plot", xlab="Improved",
                                                ylab="Frequency")

> plot(Arthritis$Improved, horiz=TRUE, main="Horizontal Bar Plot",
                        xlab="Frequency", ylab="Improved")
```

- will generate the same bar plots, but without the need to tabulate values with the table() function.

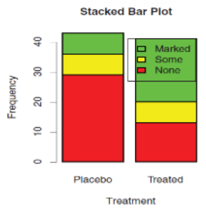
Stacked and grouped bar plots

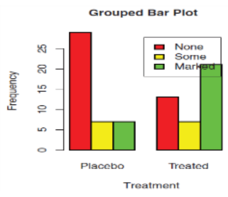
- If height is a matrix rather than a vector, the resulting graph will be a stacked or grouped bar plot.
- If beside=FALSE (the default), then each column of the matrix produces a bar in the plot, with the values in the column giving the heights of stacked "sub-bars."
- If beside=TRUE, each column of the matrix represents a group, and the values in each column are juxtaposed rather than stacked.
- Consider the cross-tabulation of treatment type and improvement status:

```
> counts <- table(Arthritis$Improved, Arthritis$Treatment)
> counts
         Treatment
Improved Placebo Treated
None     29      13
Some     7       7
Marked   7       21
```

Stacked and grouped bar plots

```
> barplot(counts, main="Stacked Bar Plot", xlab="Treatment", ylab="Frequency",
                  col=c("red", "yellow","green"), legend=rownames(counts))

> barplot(counts, main="Grouped Bar Plot", xlab="Treatment", ylab="Frequency",
                              col=c("red", "yellow", "green"),
                              legend=rownames(counts), beside=TRUE)
```





Mean bar plots

- We can create bar plots that represent means, medians, standard deviations, and so forth by using the aggregate function and passing the results to the barplot() function.

```
> states <- data.frame(state.region, state.x77)
> means <- aggregate(states$Illiteracy,
                     by=list(state.region), FUN=mean)
> means
 Group.1          x
1       Northeast     1.00
2       South         1.74
3       North Central 0.70
4       West          1.02
```

Mean bar plots

```
> means <- means[order(means$x),]
> means
      Group.1           x
3     North Central     0.70
1     Northeast     1.00
4     West          1.02
2     South         1.74

> barplot(means$x,
              names.arg=means$Group.1)
> title("Mean Illiteracy Rate")
```

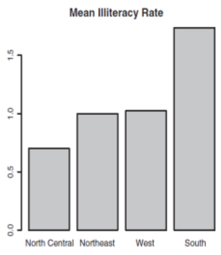


Figure 6.3 Bar plot of mean illiteracy rates for US regions sorted by rate

Tweaking bar plots

- There are several ways to tweak the appearance of a bar plot.
  - For example, with many bars, bar labels may start to overlap. You can decrease the font size using the cex.names option.
  - Specifying values smaller than 1 will shrink the size of the labels.
- Optionally, the names.arg argument allows you to specify a character vector of names used to label the bars.
- You can also use graphical parameters to help text spacing.

```
> par(mar=c(5,8,4,2))
> par(las=2)
> counts <- table(Arthritis$Improved)
```

Tweaking bar plots

```
> barplot(counts, main="Treatment Outcome", horiz=TRUE,
              cex.names=0.8, names.arg=c("No Improvement",
              "Some Improvement",  "Marked Improvement"))
```

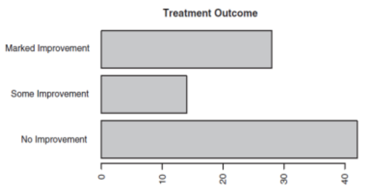


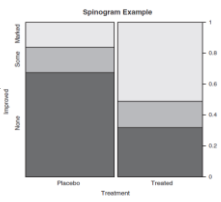
Figure 6.4 Horizontal bar plot with tweaked labels

Spinograms

- let's take a look at a specialized version called a spinogram.
- In a spinogram, a stacked bar plot is rescaled so that the height of each bar is 1 and the segment heights represent proportions.
- Spinograms are created through the spine() function of the vcd package.

```
> library(vcd)
> attach(Arthritis)
> counts <- table(Treatment, Improved)
> spine(counts, main="Spinogram Exampl
> detach(Arthritis)
```

- The larger percentage of patients with marked improvement in the Treated condition is quite evident when compared with the Placebo condition.

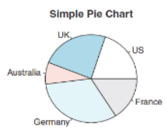


Pie Charts

- Pie charts are a popular vehicle for displaying the distribution of a categorical variable by volume.
- Pie charts are created with the function

  pie(*x*, *labels*)
- where *x* is a non-negative numeric vector indicating the area of each slice and *labels* provides a character vector of slice labels.

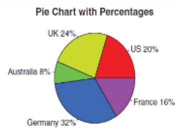
```
> par(mfrow=c(2, 2))
> slices <- c(10, 12,4, 16, 8)
> lbls <- c("US", "UK", "Australia",
    "Germany", "France")
> pie(slices, labels = lbls,
    main="Simple Pie Chart")
```



Pie Charts

- Convert the sample sizes to percentages and add the information to the slice labels.
- The second pie chart also defines the colors of the slices using the rainbow() function.
- Here rainbow(length(lbls2)) resolves to rainbow(5), providing five colors for the graph.

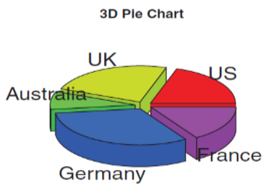
```
> pct <- round(slices/sum(slices)*100)
> lbls2 <- paste(lbls, " ", pct, "%", se
> pie(slices, labels=lbls2,
      col=rainbow(length(lbls2)),
      main="Pie Chart with Percentages")
```



Pie Charts

- The third pie chart is a 3D chart created using the pie3D() function from the plotrix package.

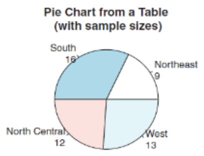
```
> library(plotrix)
> pie3D(slices, labels=lbls,explode=0.1, main="3D Pie Chart ")
```



Pie Charts

- The fourth pie chart demonstrates creation of a chart from a table.
- In this case, we count the number of states by US region and append the information to the labels before producing the plot.

```
> mytable <- table(state.region)
> lbls3 <- paste(names(mytable), "\n", mytable, sep="")
> pie(mytable, labels = lbls3,
      main="Pie Chart from a Table\n
         (with sample sizes)")
```



Fan Plot

- Pie charts make it difficult to compare the values of the slices (unless the values are appended to the labels).
  - For example, looking at the simple pie chart, can you tell how the US compares to Germany?
- In an attempt to improve on this situation, a variation of the pie chart, called a fan plot, has been developed.
- The fan plot provides you with a way to display both relative quantities and differences.
- In R, it's implemented through the fan.plot() function in the plotrix package.
- In a fan plot, the slices are rearranged to overlap each other, and the radii are modified so that each slice is visible.

Fan Plot

- Here you can see that Germany is the largest slice and that the US slice is roughly 60% as large.
- France appears to be half as large as Germany and twice as large as Australia. Remember that the *width* of the slice and not the radius is what's important here.
- It's much easier to determine the relative sizes of the slice in a fan plot than in a pie chart.

```
> library(plotrix)
> slices <- c(10, 12,4, 16, 8)
> lbls <- c("US", "UK", "Australia",
      "Germany", "France")
> fan.plot(slices, labels = lbls,
      main="Fan Plot")
```

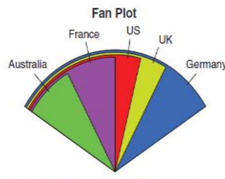


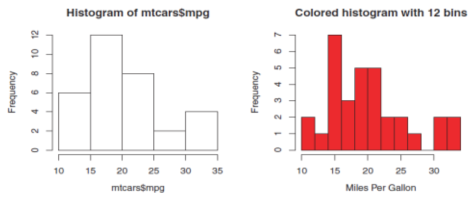
Figure 6.7 A fan plot of the country data

Histograms

- Histograms display the distribution of a continuous variable by dividing the range of scores into a specified number of bins on the x-axis and displaying the frequency of scores in each bin on the y-axis.
- We can create histograms with the function hist(x)
  - where x is a numeric vector of values.
  - The option freq=FALSE creates a plot based on probability densities rather than frequencies.
  - The breaks option controls the number of bins.
- The default produces equally spaced breaks when defining the cells of the histogram.
- The first histogram demonstrates the default plot when no options are specified. In this case, five bins are created, and the default axis labels and titles are printed.
- For the second diagram we specified 12 bins , a red fill for the bars and more attractive and informative labels and title.

Histograms

```
> hist(mtcars$mpg)

> hist(mtcars$mpg, breaks=12, col="red", xlab="Miles Per
Gallon", main="Colored histogram with 12 bins")
```



Histograms

- The third histogram maintains the same colors, bins, labels, and titles as the previous plot but adds a density curve and rug-plot overlay.
- The density curve is a kernel density estimate and is described in the next section. It provides a smoother description of the distribution of scores.
- We use the lines() function to overlay this curve in a blue color and a width that's twice the default thickness for lines.
- Finally, a rug plot is a one-dimensional representation of the actual data values.
- If there are many tied values, you can jitter the data on the rug plot using code like the following:

```
    > rug(jitter(mtcars$mpag, amount=0.01))
```

- This adds a small random value to each data point (a uniform random variate between ±amount), in order to avoid overlapping points.

Histograms

- Diagram3:

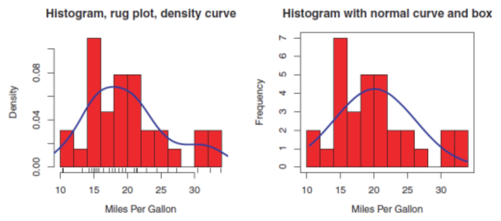
```
> hist(mtcars$mpg, freq=FALSE, breaks=12, col="red",
              xlab="Miles Per Gallon",
              main="Histogram, rug plot, density curve")
> rug(jitter(mtcars$mpg))
> lines(density(mtcars$mpg), col="blue", lwd=2)
```

- Diagram4:

```
> x <- mtcars$mpg
> h<-hist(x, breaks=12, col="red", xlab="Miles Per Gallon",
              main="Histogram with normal curve and box")
> xfit<-seq(min(x), max(x), length=40)
> yfit<-dnorm(xfit, mean=mean(x), sd=sd(x))
> yfit <- yfit*diff(h$mids[1:2])*length(x)
> lines(xfit, yfit, col="blue", lwd=2)
> box()
```

Histograms

- The fourth histogram is similar to the second but has a superimposed normal curve and a box around the figure.
- The surrounding box is produced by the box() function.



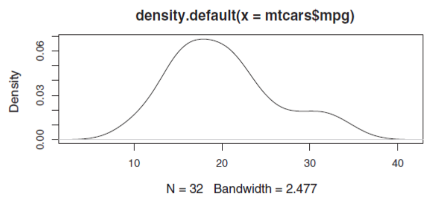
Kernel density plots

- Kernel density estimation is a nonparametric method for estimating the probability density function of a random variable.
- Although the mathematics are beyond the scope of this text, in general, kernel density plots can be an effective way to view the distribution of a continuous variable.
- The format for a density plot (that's not being superimposed on another graph) is

  plot(density(x))
  - where x is a numeric vector.
- Because the plot() function begins a new graph, use the lines() function when superimposing a density curve on an existing graph.

Kernel density plots

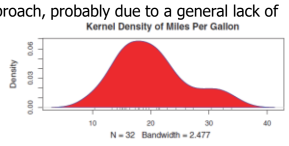
```
> par(mfrow=c(2,1))
> d <- density(mtcars$mpg)
> plot(d)
```



Kernel density plots

```
> d <- density(mtcars$mpg)
> plot(d, main="Kernel Density of Miles Per Gallon")
> polygon(d, col="red", border="blue")
> rug(mtcars$mpg, col="brown")
```

- The polygon() function draws a polygon whose vertices are given by x and y. These values are provided by the density() function in this case.
- Kernel density plots can be used to compare groups.
- This is a highly underutilized approach, probably due to a general lack of easily accessible software.
- Fortunately, the sm package fills this gap nicely.



Kernel density plots

- The sm.density.compare() function in the sm package allows you to superimpose the kernel density plots of two or more groups.
- The format is

  ```
  > sm.density.compare(x, factor)
  ```
- where x is a numeric vector and factor is a grouping variable.
- Be sure to install the sm package before first use.
  - An example comparing the mpg of cars with four, six, and eight cylinders is provided in the following diagram.

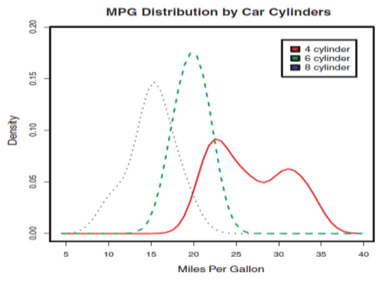
Kernel density plots

```
> library(sm)
> attach(mtcars)
> cyl.f <- factor(cyl, levels= c(4,6,8),
      labels = c("4 cylinder","6 cylinder","8 cylinder"))
> sm.density.compare(mpg, cyl, xlab="Miles Per Gallon")
> title(main="MPG Distribution by Car Cylinders")
> colfill<-c(2:(1+length(levels(cyl.f))))
> legend(locator(1), levels(cyl.f), fill=colfill)
> detach(mtcars)
```

Kernel density plots

- First, the sm package is loaded and the mtcars data frame is attached.
- In the mtcars data frame, the variable cyl is a numeric variable coded 4, 6, or 8. cyl is transformed into a factor named cyl.f, in order to provide value labels for the plot.
- The sm.density.compare() function creates the plot, and a title() statement adds a main title.
- Finally, add a legend to improve interpretability. A vector of colors is created; here, colfill is c(2,3,4). Then the legend is added to the plot via the legend() function.
- The locator(1) option indicates that you'll place the legend interactively by clicking in the graph where you want the legend to appear.
- The second option provides a character vector of the labels.
- The third option assigns a color from the vector colfill to each level of cyl.f. The results are displayed in the next figure.

Kernel density plots



Kernel density plots

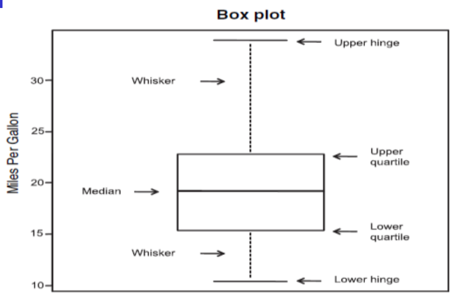
- Overlapping kernel density plots can be a powerful way to compare groups of observations on an outcome variable.
- Here you can see both the shapes of the distribution of scores for each group and the amount of overlap between groups.
- The moral of the story is that my next car will have four cylinders—or a battery.

Box plots

- A box-and-whiskers plot describes the distribution of a continuous variable by plotting its five-number summary: the minimum, lower quartile (25th percentile), median (50th percentile), upper quartile (75th percentile), and maximum.
- It can also display observations that may be outliers (values outside the range of ± 1.5*IQR, where IQR is the interquartile range defined as the upper quartile minus the lower quartile).
- For example, this statement produces the plot shown in figure 6.11:

```
> boxplot(mtcars$mpg, main="Box plot",
                          ylab="Miles per Gallon")
```

Box plots



Box plots

- The diagram include annotations added by hand to illustrate the components.
- Each whisker extends to the most extreme data point, which is no more than 1.5 times the interquartile range for the box. Values outside this range are depicted as dots.
  - For example, in the sample of cars, the median mpg is 19.2, 50% of the scores fall between 15.3 and 22.8, the smallest value is 10.4, and the largest value is 33.9.
- How did I read this so precisely from the graph?
- Issuing boxplot.stats(mtcars$mpg) prints the statistics used to build the graph.
- There don't appear to be any outliers, and there is a mild positive skew (the upper whisker is longer than the lower whisker).

Using parallel box plots to compare groups

- Box plots can be created for individual variables or for variables by group.
- The format is

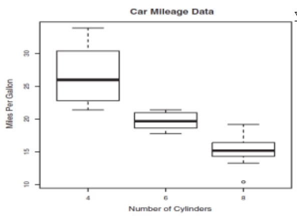
```
> boxplot(formula, data=dataframe)
```

  - where *formula* is a formula and *dataframe* denotes the data frame (or list) providing the data.
  - An example of a formula is y ~ A, where a separate box plot for numeric variable y is generated for each value of categorical variable A.
  - The formula y ~ A*B would produce a box plot of numeric variable y, for each combination of levels in categorical variables A and B.
- Adding the option varwidth=TRUE makes the box-plot widths proportional to the square root of their sample sizes.
- Add horizontal=TRUE to reverse the axis orientation.

Using parallel box plots to compare groups

- The following code revisits the impact of four, six, and eight cylinders on auto mpg with parallel box plots.

```
> boxplot(mpg ~ cyl, data=mtcars, main="Car Mileage
              Data", xlab="Number of Cylinders",
                                ylab="Miles Per Gallon")
```



Using parallel box plots to compare groups

- You can see that there’s a good separation of groups based on gas mileage.
- We can also see that the distribution of mpg for six-cylinder cars is more symmetrical than for the other two car types.
- Cars with four cylinders show the greatest spread (and positive skew) of mpg scores, when compared with six- and eight cylinder cars.
- There’s also an outlier in the eight-cylinder group.

Notched Box Plots

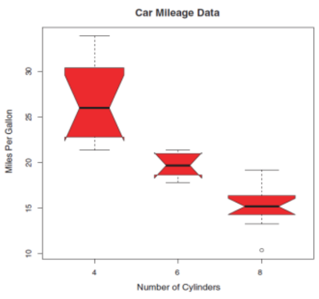
- Box plots are very versatile.
- By adding notch=TRUE, we get notched box plots.
- If two boxes’ notches don’t overlap, there’s strong evidence that their medians differ.
- The following code creates notched box plots for the mpg example:

```
> boxplot(mpg ~ cyl, data=mtcars, notch=TRUE,
      varwidth=TRUE, col="red", main="Car Mileage
      Data", xlab="Number of Cylinders",
      ylab="Miles Per Gallon")
```

- The col option fills the box plots with a red color, and varwidth=TRUE produces box plots with widths that are proportional to their sample sizes.

Notched Box Plots

- You can see that the median car mileage for four-, six-, and eight-cylinder cars differs.
- Mileage clearly decreases with number of cylinders.



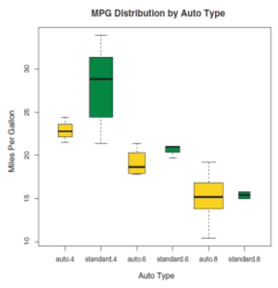
Box Plots for more than one grouping factor

- The following provides box plots for mpg versus the number of cylinders and transmission type in an automobile.
- We can use the col option to fill the box plots with color.
- Note that colors recycle; in this case, there are six box plots and only two specified colors, so the colors repeat three times.

```
> mtcars$cyl.f<- factor(mtcars$cyl,
      levels=c(4,6,8), labels=c("4","6","8"))
> mtcars$am.f <- factor(mtcars$am, levels=c(0,1),
                  labels=c("auto", "standard"))
> boxplot(mpg ~ am.f *cyl.f, data=mtcars,
      varwidth=TRUE, col=c("gold","darkgreen"),
      main="MPG Distribution by Auto Type",
      xlab="Auto Type", ylab="Miles Per Gallon")
```

Box Plots for more than one grouping factor

- It’s again clear that median mileage decreases with number of cylinders.
- For 4- and 6-cylinder cars, mileage is higher for standard transmissions.
- But for 8-cylinder cars, there doesn’t appear to be a difference.
- You can also see from the widths of the box plots that standard 4-cylinder and automatic 8-cylinder cars are the most common in this dataset.



Violin plots

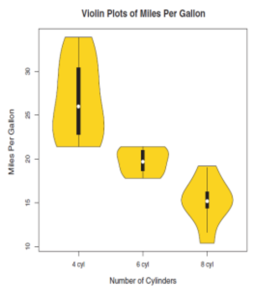
- A violin plot is a combination of a box plot and a kernel density plot.
- We can create it by using vioplot() function from the vioplot package. Be sure to install the vioplot package before first use.
- The format for the vioplot() function is

```
> vioplot(x1, x2, ... , names=, col=)
```

  - where *x1*, *x2*, … represent one or more numeric vectors to be plotted (one violin plot is produced for each vector).
- The *names* parameter provides a character vector of labels for the violin plots, and *col* is a vector specifying the colors for each violin plot.

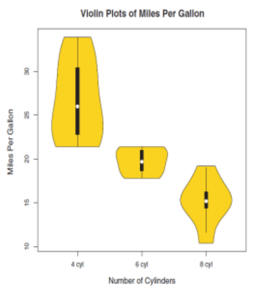
Violin plots

```
> library(vioplot)
> x1 <- mtcars$mpg[mtcars$cyl==4]
> x2 <- mtcars$mpg[mtcars$cyl==6]
> x3 <- mtcars$mpg[mtcars$cyl==8]

> vioplot(x1, x2, x3,
names=c("4 cyl", "6 cyl", "8 cyl"),
col="gold")

> title("Violin Plots of Miles Per
Gallon", ylab="Miles Per Gallon",
xlab="Number of Cylinders")
```



Violin plots

- Note that the vioplot() function requires to separate the groups to be plotted into separate variables.
- Violin plots are basically kernel density plots superimposed in a mirror-image fashion over box plots.
- Here, the white dot is the median, the black boxes range from the lower to the upper quartile, and the thin black lines represent the whiskers.
- The outer shape provides the kernel density plot.



Dot plots

- Dot plots provide a method of plotting a large number of labeled values on a simple horizontal scale.
- You create them with the dotchart() function, using the format

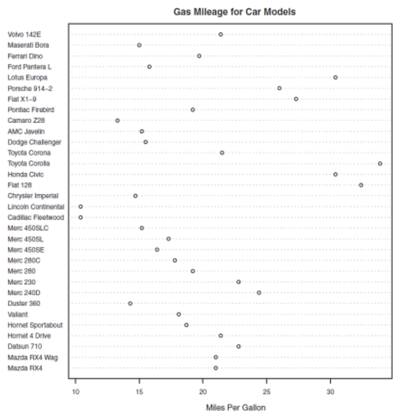
  ```
  > dotchart(x, labels=)
  ```

  - where x is a numeric vector and labels specifies a vector that labels each point.
- We can add a groups option to designate a factor specifying how the elements of x are grouped.
- If so, the option gcolor controls the color of the groups label, and cex controls the size of the labels.
- Here's an example with the mtcars dataset:

```
> dotchart(mtcars$mpg, labels=row.names(mtcars), cex=.7,
             main="Gas Mileage for Car Models",
             xlab="Miles Per Gallon")
```

Dot p

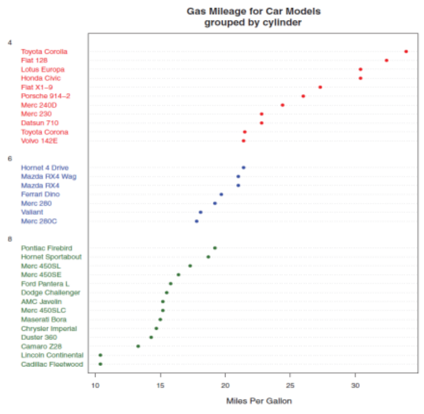
- This graph allows you to see the mpg for each make of car on the same horizontal axis.



Dot plots

- Dot plots typically become most interesting when they're sorted and grouping factors are distinguished by symbol and color.

```
> x <- mtcars[order(mtcars$mpg),]
> x$cyl <- factor(x$cyl)
> x$color[x$cyl==4] <- "red"
> x$color[x$cyl==6] <- "blue"
> x$color[x$cyl==8] <- "darkgreen‟

> dotchart(x$mpg, labels = row.names(x), cex=.7,
       groups = x$cyl, gcolor = "black", color = x$color,
       pch=19, xlab = "Miles Per Gallon”,
       main = "Mileage for Car Models grouped by cylinder")
```



Dot plots

- A number of features become evident for the first time.
- Again, we see an increase in gas mileage as the number of cylinders decreases.
  - For example, the Pontiac Firebird, with 8 cylinders, gets higher gas mileage than the Mercury 280C and the Valiant, each with 6 cylinders.
  - The Hornet 4 Drive, with 6 cylinders, gets the same miles per gallon as the Volvo 142E, which has 4 cylinders.
  - It's also clear that the Toyota Corolla gets the best gas mileage by far, whereas the Lincoln Continental and Cadillac Fleetwood are outliers on the low end.
- We can gain significant insight from a dot plot in this example because each point is labeled, the value of each point is inherently meaningful, and the points are arranged in a manner that promotes comparisons.
- But as the number of data points increases, the utility of the dot plot decreases.