MTH 2520 R Notes 11

10 Graphics

- R's built-in graphics functions can be split into three categories:
  - ***High-level*** plotting functions are used to *create* plots.
  - ***Low-level*** plotting functions are used to *add* text, lines, points etc. *to existing* plots.
  - ***Interactive*** plotting functions are used to *interact* with plots via mouse clicks in the graphics window.

10.1 Creating Plots

- Here are some of the most commonly used *high-level* functions for creating plots:

```
plot()              # Scatterplot, time-series plot
hist()              # Histogram
boxplot()           # Boxplot(s)
curve()             # Graph of a function
barplot()           # Bar chart of specified bar heights
pie()               # Pie chart of specified pie areas
stripchart()        # Dot plot, individual value plot
pairs()             # Scatterplot matrix
qqnorm()            # Normal probability plot (quantile-quantile plot)
stem()              # Stem and leaf plot
```

10.1.1 Scatterplots

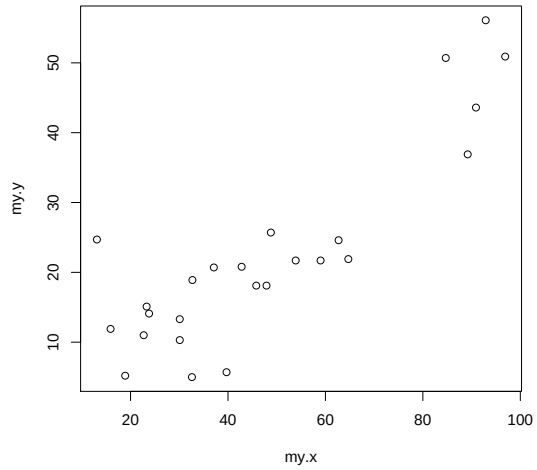
- `plot()` takes arguments `x` and `y` and plots them in a ***scatterplot***. Some other arguments that can be passed to `plot()` are:

```
main            # Main title (in quotation marks)
sub             # Subtitle (in quotation marks)
xlab, ylab      # Labels for the x and y axes (in quotation marks)
xlim, ylim      # Limits for the x and y axes in the plot (in the form
                # c(lower, upper) )
type            # Type of plot that should be drawn (e.g. points, lines,
                # etc.)
...             # Other arguments such as the graphical parameters that
```

```
# can be controlled by par()
```

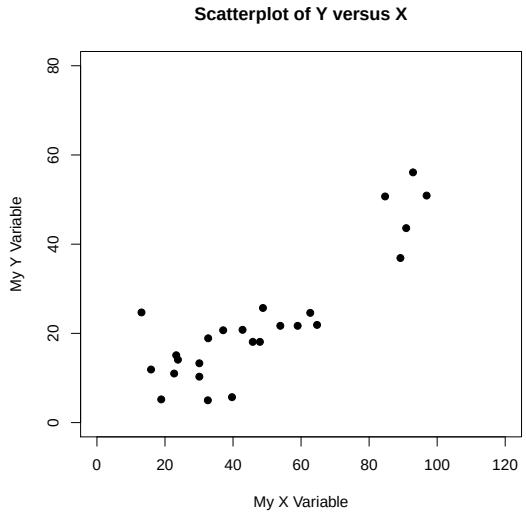
- Here's an example using these vectors `my.x` and `my.y`:

```
> my.x

 [1] 18.9 53.9 42.8 47.9 37.1 23.3 90.9 30.1 96.9 22.7 15.9 45.8 59.0 89.2 30.1 92.9 32.6 84
[19] 64.7 48.8 23.8 62.7 13.1 39.7 32.7

> my.y

 [1]  5.2 21.7 20.8 18.1 20.7 15.1 43.6 10.3 50.9 11.0 11.9 18.1 21.7 36.9 13.3 56.1  5.0 50
[19] 21.9 25.7 14.1 24.6 24.7  5.7 18.9

> plot(x = my.x, y = my.y)
```



- Here's a nicer version of the plot:

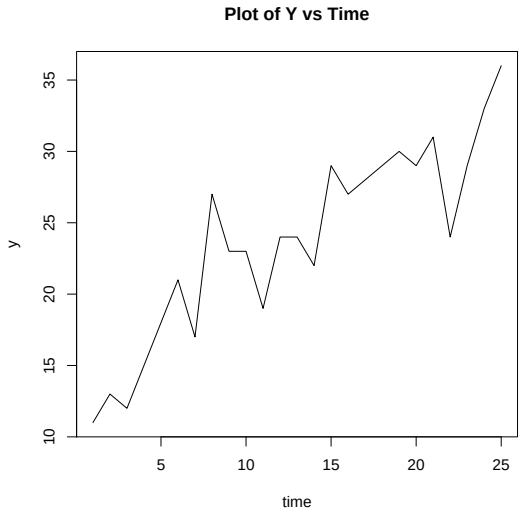
```
> plot(x = my.x, y = my.y,
+      main = "Scatterplot of Y versus X",
+      xlab = "My X Variable",
+      ylab = "My Y Variable",
+      xlim = c(0, 120),
+      ylim = c(0, 80),
+      pch = 19)
```



(The argument `pch`, for "plot character", is one of the '`...`' arguments that can be passed to `plot()` and that can also be set by the `par()` function. Specifying `pch=19` indicates solid circles for the points. To see a list of the available point types, look for `pch` in the help file for `par()`.)

- In a ***time-series plot***, x represents time and the points are connected by lines. To make one, we specify `type="l"` (the letter `"l"` for "line") in `plot()`. For example:

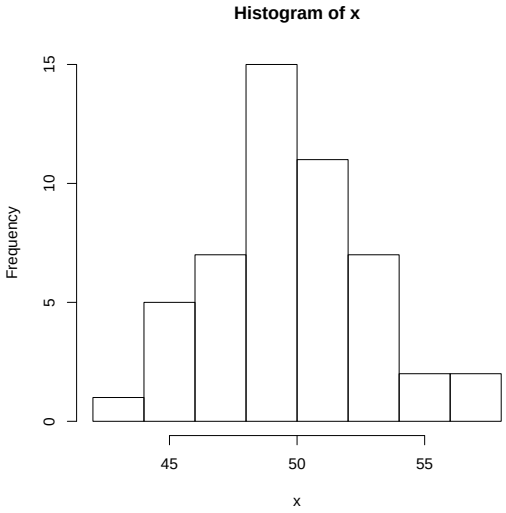
```
> y

 [1] 11 13 12 15 18 21 17 27 23 23 19 24 24 22 29 27 28 29 30 29 31 24 29 33 36

> time

 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25

> plot(x = time, y = y, type = "l", main = "Plot of Y vs Time")
```



10.1.2 Histograms and Boxplots

- `hist()` takes a vector argument `x` and produces a histogram of the data. For example:

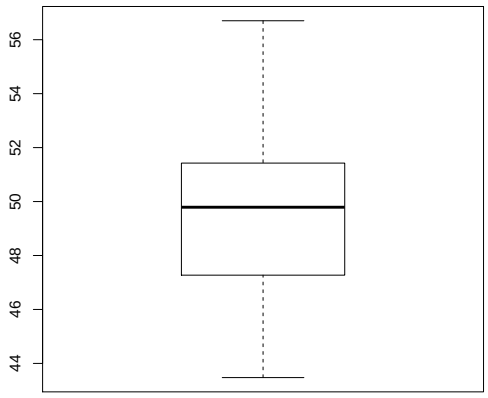
```
> x <- rnorm(n = 50, mean = 50, sd = 4)
> hist(x)
```



(The `rnorm()` function generates a random sample from a *normal* distribution with a specified mean and standard deviation.)

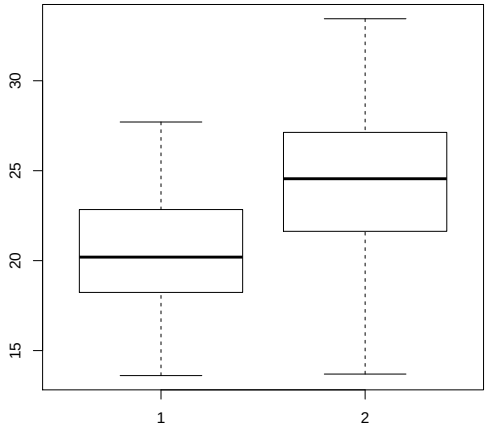
- `boxplot()` takes one or more vector arguments and produces the boxplot(s). For example:

```
> boxplot(x)
```



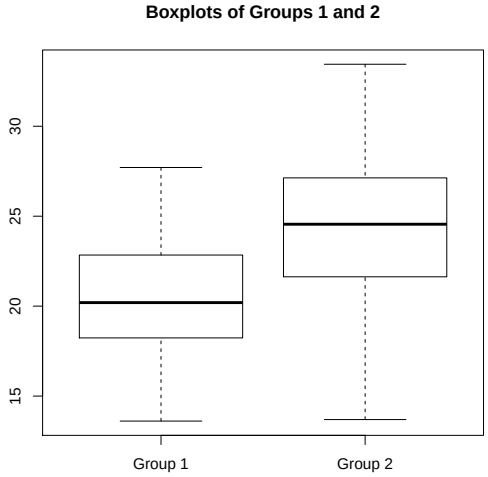
- **Interpretation of a Histogram**:
  - The range of the data (smallest value to largest) is partitioned into intervals, or ***bins***, which correspond to the bases of the bars in the histogram.
  - The number of data values falling into a bin (i.e. the ***frequency*** for that bin) determines the height of the bar over the bin.
- **Interpretation of a Boxplot**:
  - Data values are represented along the vertical axis.
  - The top of the box is at the ***third quartile***, so 75% of the data values lie below the top of the box and 25% lie above it.
  - The bottom of the box is at the ***first quartile***, so 25% of the data values lie below the bottom of the box and 75% lie above it.
  - The horizontal line through the box is at the ***median***, so half of the data values lie below that line and half lie above it.
  - The "whiskers" (vertical lines) extend above and below the box to the largest and smallest data values, unless those values are *outliers*, in which case the whiskers only extend to the largest and smallest values that aren't *outliers*.
- To produce side-by-side boxplots, we pass multiple vectors to `boxplot()`. For example:

```
> x1 <- rnorm(n = 50, mean = 20, sd = 3)
> x2 <- rnorm(n = 40, mean = 25, sd = 4)

> boxplot(x1, x2)
```



- We can include labels below the boxes via the `names` argument (and add a title via `main`):

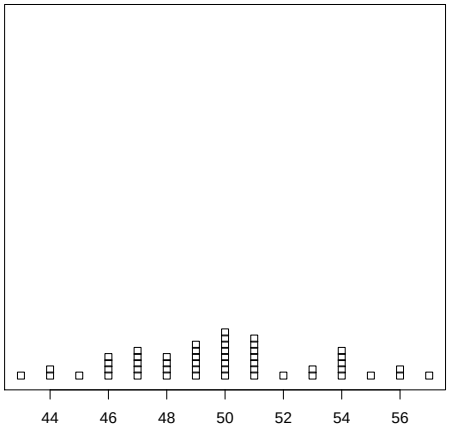
```
> boxplot(x1, x2, names = c("Group 1", "Group 2"),
+     main = "Boxplots of Groups 1 and 2")
```



10.1.3 Dot Plots

- The function `stripchart()` will produce a dot plot of a data set if we specify `method = "stack"`. It's sometimes necessary to round the data values to get them to stack on top of each other:

```
> x <- round(x)
> stripchart(x, method = "stack", at = 0)
```

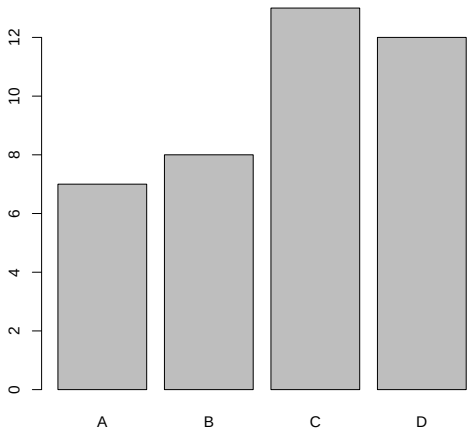


Specifying `at=0` indicates that we want base of the stacks of dots to be "at" the horizontal axis ($y = 0$).

10.1.4 Bar Plots and Pie Charts

- ***Qualitative*** (or ***categorical***) data are usually displayed in bar plots or pie charts.
- `barplot()` takes a vector argument `height` containing bar heights and produces the bar plot. For example:

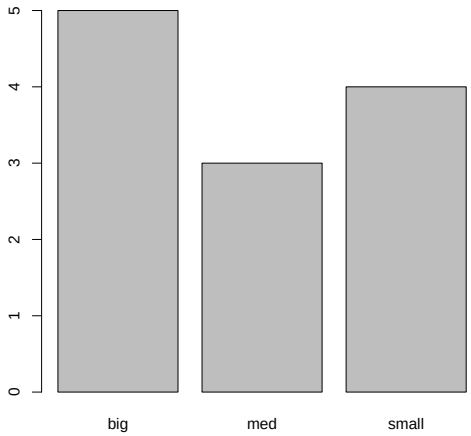
```
> bar.hts <- c(7, 8, 13, 12)

> barplot(height = bar.hts, names.arg = c("A", "B", "C", "D"))
```



(The `names.arg` argument was used to the add labels below the bars.)

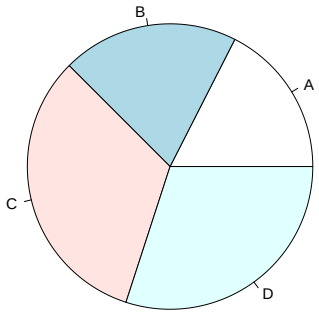
- The bar heights can be obtained from a `"character"` vector or factor using `table()`, and a *table* object can be passed directly to `barplot()`:

```
> char.vec <- c("big", "big", "med", "small", "med", "big", "big", "small",
+               "small",  "med", "big", "small")
> my.tab <- table(char.vec)
> barplot(my.tab)
```



- `pie()` takes a vector argument `x` indicating the *relative* sizes of the pie slices, and produces a pie chart. For example:

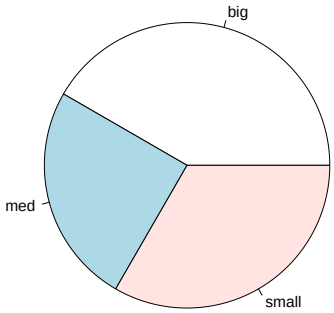
```
> slice.sizes <- c(7, 8, 13, 12)

> pie(x = slice.sizes, labels = c("A", "B", "C", "D"))
```



(The `labels` argument was used to the add labels to the slices.)

- The pie areas can be obtained from a `"character"` vector or factor using `table()`, and a *table* object can be passed directly to `pie()`. For example (using `char.vec` from above):

```
> my.tab <- table(char.vec)
> pie(my.tab)
```



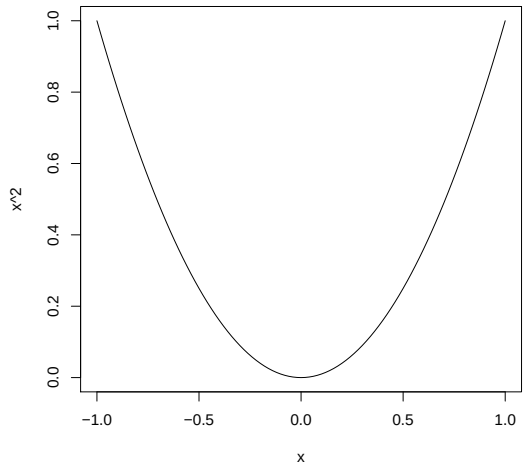
10.1.5 Graphing a Curve

- There are two ways to graph a curve in R:
  - Using `curve()`.
  - Using `plot()`, specifying `type="l"` (for "line").
- `curve()` takes as its main argument either:
  - An expression involving a variable `x` and representing a mathematical function, (e.g. `x^2` or `1/x`),

    or
  - The name of an existing function in R (e.g. `log` or `sqrt`).

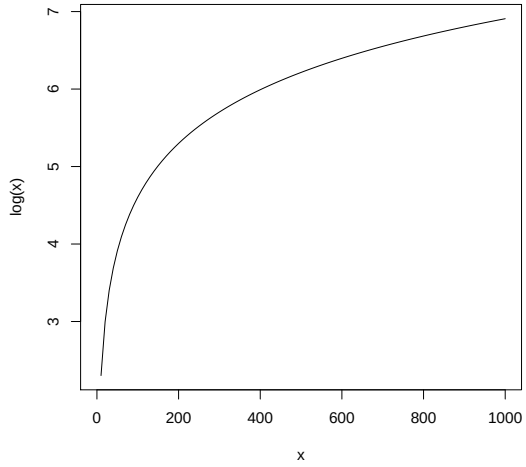
  It graphs the curve over an interval specified by the arguments `from` and `to`.
- For example, to graph $f(x) = x^2$ over the interval from -1 to 1, type:

```
> curve(x^2, from = -1, to = 1)
```



and to graph the natural logarithmic function (`log()` in R), type:

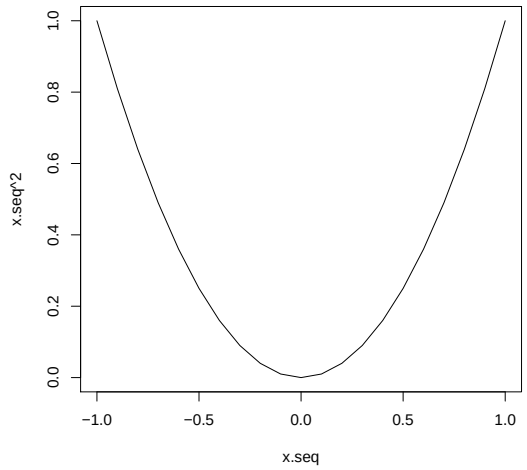
```
> curve(log, from = 0, to = 1000)
```



- To graph a curve using `plot()`, pass to `plot()` a a vector containing a sequence of values for `x` and an expression involving that vector, representing a mathematical function (e.g. $f(x) = x^2$ or $f(x) = 1/x$), and specify `type="l"` (for "line"):
- For example, to graph $f(x) = x^2$ from -1 to 1, type:

```
> x.seq <- seq(from = -1, to = 1, by = 0.1)
```

```
> plot(x = x.seq, y = x.seq^2, type = "l")
```



Section 10.1 Exercises

Exercise 1 The data set `state.x77` comes built-in with R. The vectors `illit` and `murder`, created below, contain illiteracy and murder rates for each of the 50 states in the U.S.:

```
> illit <- state.x77[ , 3]
> murder <- state.x77[ , 5]
```

Use `plot()` to make a scatterplot of the murder rates (y-axis) versus illiteracy rates (x-axis). Use the arguments `main`, `xlab`, `ylab`, `xlim`, and `ylim` to modify the main title, x and y axis labels, and x and y axis limits. Report your R command.

Exercise 2 Most plotting functions exclude `NA`s from the plot. In particular no point will be plotted by `plot()` if one or both of its coordinates are `NA`.

How many points will appear in a `plot()` of the following vectors? Check your answer.

```
> x <- c(1, 2, 3, NA, 5)
> y <- c(3, 7, 6, 8, 11)
```

Exercise 3 Recall that `table()` takes a *factor* or `"character"` vector argument and returns a *table* of counts.

Both `barplot()` and `pie()` accept *tables* as arguments, and produce plots from the table counts.

A recent Gallup poll asked people if they smiled or laughed "a lot" in a given day. Here's a representative sampling of the responses:

```
> laughed <- c("Yes", "Yes", "Yes", "No", "Yes", "No", "Yes", "Yes", "Yes",
               "Yes", "No", "Yes", "Yes", "Yes", "No", "No", "Yes", "Yes",
               "Yes", "No", "Yes")
```

a) Use `table()` to create a *table* from the `"character"` vector `laughed`, then pass the table to `barplot()` to make a bar plot of the table counts. Report your R command(s).

b) Now pass the *table* to `pie()` to make a pie chart of the counts. Report your R command(s).

Exercise 4 Graph the polynomial function

$$f(x) = 1 - 2x + x^2 + 3x^3$$

over the interval from -2 to 2 using two different methods, and report your R command(s):

a) Use `curve()`, with `from=-2` and `to=2`.

b) Use `plot()`, with `type="l"`, after using `seq()` to create a sequence of x values from -2 to 2.

10.2 Customizing Plots

10.2.1 Setting Graphical Parameters Using `par()`

- A number of plot features (***graphical parameters***) can be controlled using the function:

```
par()               # Get or set graphical parameters
```

- Here are just some of the graphical parameters that can be set by `par()`:

```
pch          # Plot character, or symbol type (an integer from 0 to 25)
cex          # Character expansion factor, i.e. size of plot characters
             # and/or text (values greater than 1 increase the size)
lty, lwd     # Line type (e.g. "dashed" or "solid") and line width
             # (values greater than 1 increase the width)
col          # Color of the objects being plotted (in quotation marks)
bg, fg       # Background and foreground colors (in quotation marks)
mfrow, mfcol # Multiple-figure plot arrangement (as a numerical vector
             # of the form c(nrow, ncol))
xaxt, yaxt   # x and y axis types (specify "n" for no axis)
tcl          # Length of axis tick marks (as a fraction of a line of
             # margin text)
bty          # Type of box drawn around the plot (specify "n" for none)
mar, mai     # Margin size in number of lines of margin text or inches
```

```
              # (a numerical vector of the form c(bottom, left, top,
              # right))
cex.main,
cex.axis,
cex.lab       # Character expansion factor (size of text) for main
              # title, axis annotations, and axis labels (values greater
              # than 1 increase their sizes)
```

For the full list, look at the help file for `par()`:

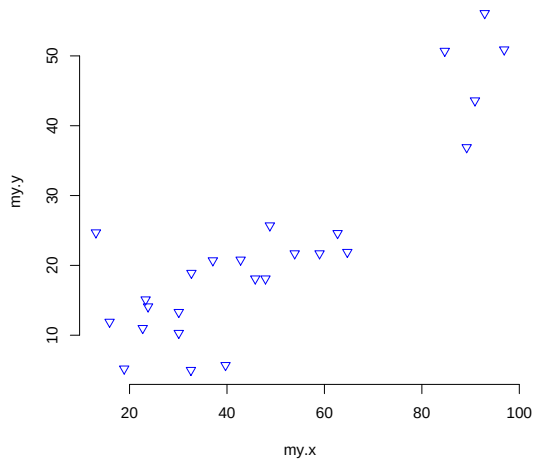
```
> ? par
```

- Setting a graphical parameter using `par()` affects *all subsequent plots* made during the current R session.
- As an example, to change plot symbols to triangles (`pch=25`) that are blue (`col="blue"`) and eliminate the box around our plots (`bty="n"`), we can type:

  ```
  > par(pch = 25, col = "blue", bty = "n")
  ```

  Now *all subsequent plots* will have these features (until we reset them using `par()` again):

  ```
  > plot(my.x, my.y)
  ```



To reset the graphical parameters back to their original settings, we type:

```
> par(pch = 1, col = "black", bty = "o")    # Returns the graphical parameters
                                            # to their original settings.
```

- To see the current settings for all of the graphical parameters, type:

```
> par()
```

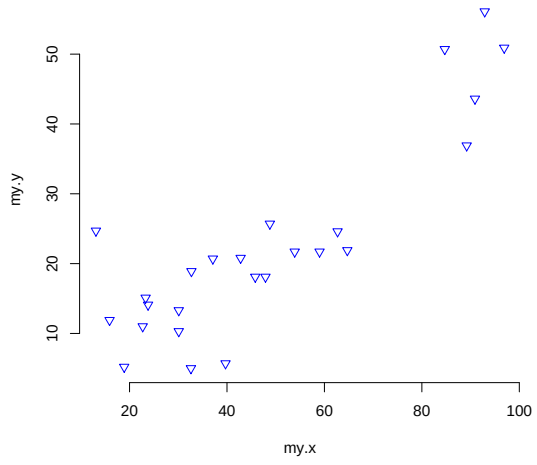
- Here's a trick that saves the original graphical parameter settings *before* `par()` changes them so they can easily be reset after the plot has been made:

```
> opar <- par(pch = 25, bty = "n")  # Saves the original graphical parameter
                                    # settings in opar before calling par().

> plot(my.x, my.y)

> par(opar)                         # Returns the graphical parameters to
                                    # their original settings.
```

10.2.2 Setting Graphical Parameters in `plot()` (and Other High-Level Plotting Functions)

- Most of `par()`'s graphical parameters can also be passed as the arguments '`...`' to `plot()` (and other high-level plotting functions, e.g. `hist()`, `barplot()`, etc.). In this case it *only affects the current plot.*
- For example, we can duplicate the scatterplot of Section 10.2.1 by indicating `pch=25`, `col="blue"`, and `bty="n"` in the call to `plot()`:

```
> plot(my.x, my.y, pch = 25, col = "blue", bty = "n")
```



(In this case, *only the current plot is affected*, so we don't need to reset `pch`, `col`, and `bty` back to their original settings after the call to `plot()`.)

Section 10.2 Exercises

Exercise 5 This problem involves `par()`.

a) Setting a graphical parameter using `par()` affects *all subsequent plots* until either the graphical parameter is reset (using `par()` again) or the R session is terminated.

What color will the points be in the *second* plot below?

```
> par(col = "red")
> plot(x, y)
> plot(x, y)                    # What color will this plot be?
> par(col = "black")
```

b) What color will the points be in the *second* plot below?

```
> opar <- par(col = "red")    # Saves the original graphical parameter
                              # settings in opar before calling par().
                              # The original color is "black".
> plot(x, y)
> par(opar)                   # Returns the graphical parameters to
                              # their original settings.

> plot(x, y)                  # What color will this plot be?
```

Exercise 6 Many of the graphical parameters that can be set by `par()` can also be passed as arguments '`...`' to `plot()`. Among them are `cex` (character expansion) and `pch` (plot character).

`cex` is represented by a numerical value, with values greater than 1 expanding the characters and values less than 1 shrinking them.

`pch` can be either a `"character"` or an integer representing a symbol. To see a list of available symbols, look for `pch` in `par()`'s argument list in the help file:

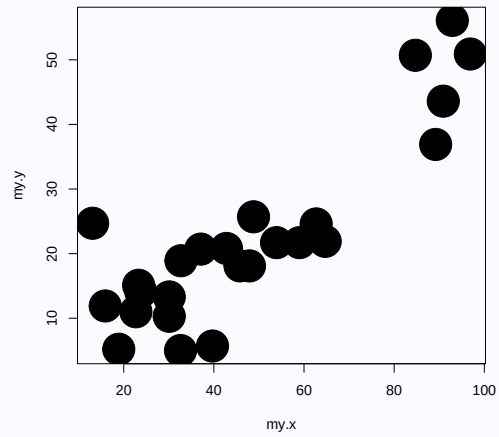
```
> ? par
```

(In the help file for `par()`, you'll be directed to the help file for `points()`.)

a) Here are two vectors

```
> my.x <- c(18.9, 53.9, 42.8, 47.9, 37.1, 23.3, 90.9, 30.1, 96.9, 22.7,
            15.9, 45.8, 59.0, 89.2, 30.1, 92.9, 32.6, 84.7, 64.7, 48.8,
            23.8, 62.7, 13.1, 39.7, 32.7)

> my.y <- c(5.2, 21.7, 20.8, 18.1, 20.7, 15.1, 43.6, 10.3, 50.9, 11.0,
            11.9, 18.1, 21.7, 36.9, 13.3, 56.1, 5.0, 50.7, 21.9, 25.7,
            14.1, 24.6, 24.7, 5.7, 18.9)
```

Make a scatterplot of the data with solid circles (`pch=19`) that are five times as large as usual (`cex=5`) by specifying these two settings in the call to `plot()`. Report your R command.



Exercise 7 Many of the graphical parameters that can be set by `par()` can also be passed as arguments '`...`' to `plot()`. Among them are `lty` (line type), `lwd` (line width), and `col` (color).

`lwd` is represented by a numerical value, with values greater than 1 widening the line, and values less than 1 narrowing it.

`lty` is used to make dashed lines, dotted lines, etc., and is represented by a `"character"` string indicating the line type. To see a list of available types, look for `lty` in `par()`'s argument list:

```
> ? par
```

`col` is represented by a `"character"` string indicating a color. To see a list of available colors, type:

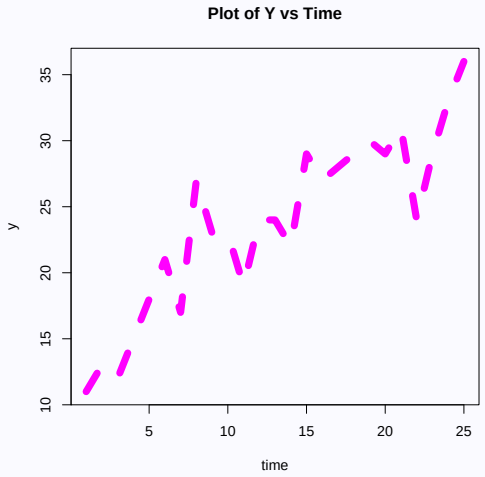
```
> colors()
```

Here are two vectors:

```
> time <- 1:25
```

```
> y <- c(11, 13, 12, 15, 18, 21, 17, 27, 23, 23, 19, 24, 24, 22, 29,
         27, 28, 29, 30, 29, 31, 24, 29, 33, 36)
```

Use `plot()`, with `type="l"` (for "line"), to make a *time-series plot* of the data in which `lty`, `lwd`, and `col` are passed as arguments to `plot()` and set to something other than their default settings. Report your R command. For example, your plot might look something like this:



Exercise 8 Many of the graphical parameters that can be set by `par()` can also be passed as arguments '`...`' to `boxplot()`. Among those graphical parameters is `col` (color).

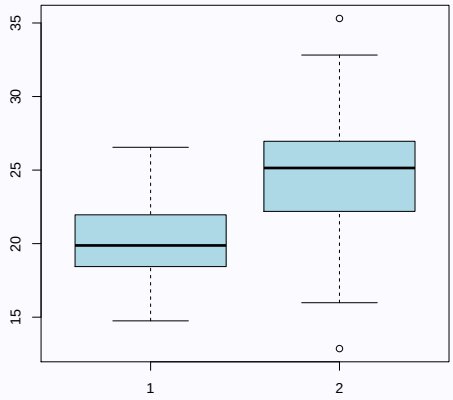
`col` is represented by a `"character"` string indicating a color. It can be represented in other ways too. To see a list of available colors, as `"character"` strings, type:

```
> colors()
```

Execute the following commands to generate two vectors `x1` and `x2`:

```
> set.seed(26)
> x1 <- rnorm(n = 50, mean = 20, sd = 3)
> x2 <- rnorm(n = 40, mean = 25, sd = 4)
```

Now make side-by-side boxplots of `x1` and `x2`, with `col` passed as an argument to `plot()` and set to something other than its default setting. (i.e. to something from the list returned by typing `colors()`). Report your R command. For example, if you specified `col="lightblue"` in `boxplot()`, your plot might look something like this:



10.3 Adding to an Existing Plot

- Here are some of the *low-level* functions for adding features to an existing plot:

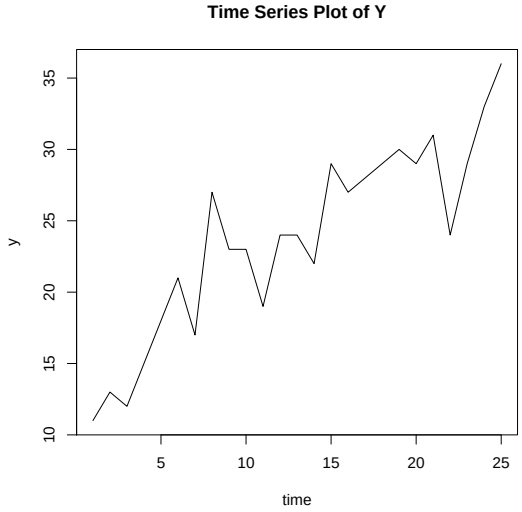
```
points()     # Add points to the plot at specified coordinates
lines()      # Add a line to the plot connecting specified coordinates
polygon()    # Draw a polygon in the plot with a given set of vertices
abline()     # Add a line to the plot with given intercept a and
             # slope b
segments()   # Add line segments to the plot between pairs of points
arrows()     # Draw an arrow in the plot (with specified start and
             # end points)
text()       # Add text to the plot at a specified set of coordinates
mtext()      # Add text in a margin of the plot
legend()     # Add a legend to the plot
title()      # Add a main title to the plot (if it doesn't already
             # have one).  Can also be used to add x and y axis labels.
axis()       # Add an axis to the plot on a given side
curve()      # Add a curve to the plot (specify add = TRUE)
qqline()     # Add a line to a normal probability plot
box()        # Add a box around the plot (if one doesn't already exist)
rect()       # Draw a rectangle in the plot at a given set of
             # coordinates
symbols()    # Add various symbols to the plot (circles, squares,
             # etc.)
```

- In addition to their main arguments (see their help files), these functions also accept arguments '`...`' representing graphical parameters that can be set by `par()` (e.g. `col`, `pch`,

`cex`, `lwd`, etc.).

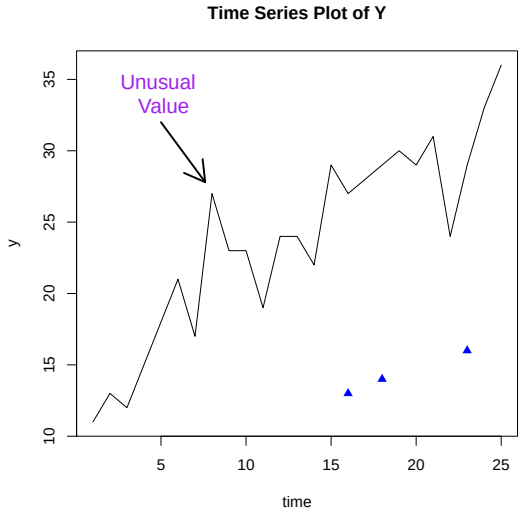
- As an example, consider the time-series plot (using `time` and `y` created earlier):

```
> plot(x = time, y = y, pch = 19, type = "l",
+      main = "Time Series Plot of Y")
```



Below, we use `text()`, `arrows()`, and `points()` to add to the plot:

```
> text(x = 5, y = 34, labels = "Unusual \n Point", cex = 1.3, col = "purple")
> arrows(x0 = 5, y0 = 32, x1 = 7.6, y1 = 27.8, lwd = 2)

> new.x <- c(16, 18, 23)
> new.y <- c(13, 14, 16)
> points(x = new.x, y = new.y, pch = 17, col = "blue")
```



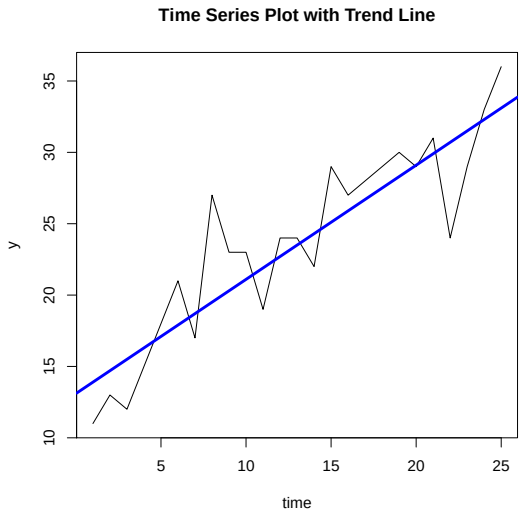
(The symbol `\n` in the call to `text()` above is a "new line" character.)

- As another example, consider the time-series plot:

```
> plot(x = time, y = y, type = "l",
       main = "Time Series Plot with Trend Line")
```

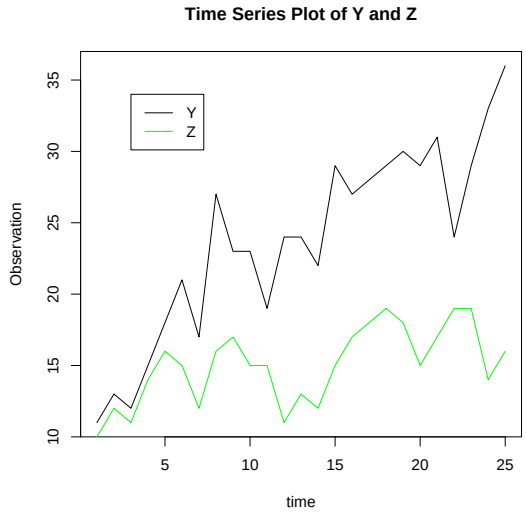
Below, we use `abline()` to add a trend line (with intercept $a = 13.1$ and slope $b = 0.8$):

```
> abline(a = 13.1, b = 0.8, lwd = 3, col = "blue")
```



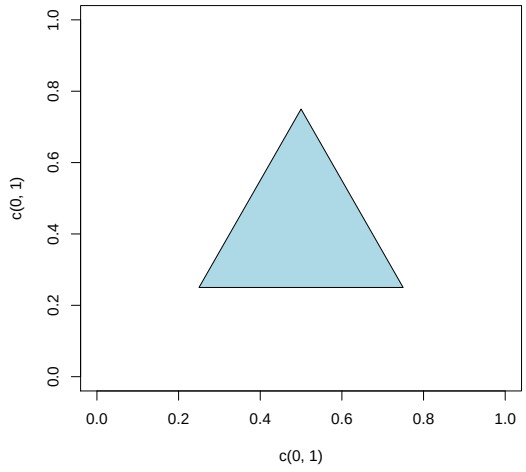
- Here's another example that uses `lines()` to add a new time-series (`z`) to an *existing* time-series plot (of `y`) and uses `legend()` to add a legend:

```
> z

 [1] 10 12 11 14 16 15 12 16 17 15 15 11 13 12 15 17 18 19 18 15 17 19 19 14 16

> plot(x = time, y = y, type = "l", ylab = "Observation",
+    main = "Time Series Plot of Y and Z")

> lines(time, z, col = "green")

> legend(x = 3, y = 34, legend = c("Y", "Z"), col = c("black", "green"),
+    lty = c("solid", "solid"))
```



- `polygon()` takes vector arguments `x` and `y` representing coordinates of a polygon's vertices, and `col` giving a fill color, and adds the polygon to the plot. Below, a blank plot is created first, and the polygon is added to the blank plot:

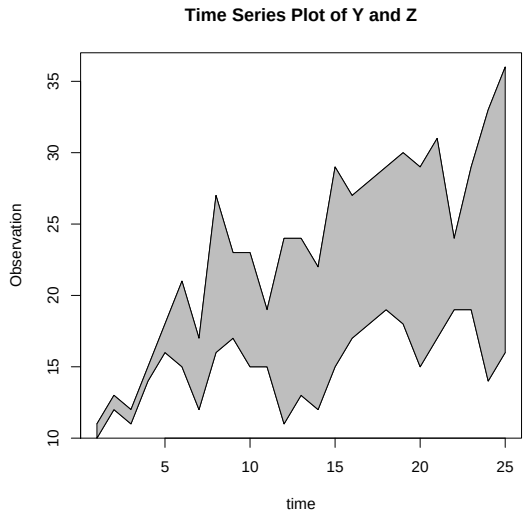
```
> plot(x = c(0, 1), y = c(0, 1),          # Just set up the coordinate system
       col = "transparent")

> polygon(x = c(0.25, 0.5, 0.75),         # Add the polygon (a triangle)
          y = c(0.25, 0.75, 0.25),
          col = "lightblue")
```



(Above, the `x` and `y` coordinates passed to `polygon()` are in clockwise order.)

- Below, `poloygon()` is used to fill in the gap between two time-series lines:

```
> plot(x = time, y = y, type = "l", ylab = "Observation",
       main = "Time Series Plot of Y and Z")
> lines(time, z)

> polygon(x = c(time, rev(time)), y = c(y, rev(z)), col = "grey"
```



(Above, the `x` and `y` coordinates passed to `polygon()` were in clockwise order.)

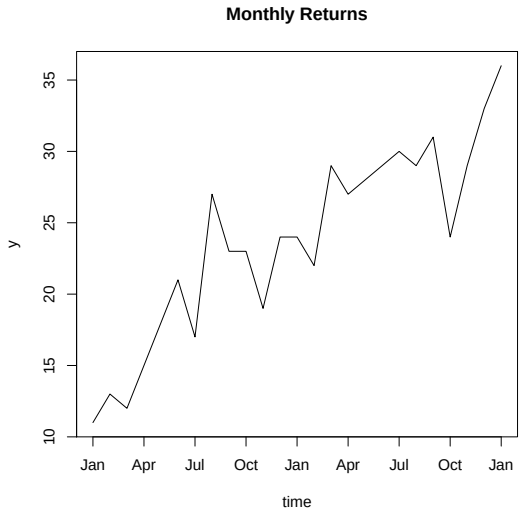
- We can control the appearance of the axes of a plot using `axis()`. To do so:

  1. First create the plot *without any axes* by specifying `xaxt="n"` and/or `yaxt="n"` ("none" for the axis type) in the call to `plot()` (or whichever high-level plotting function you're using).
  2. Then use `axis()` to add a customized axis on the appropriate side (`side=1` for the x axis, `side=2` for the y axis).

- For example, suppose the vector `y` (created earlier) contains monthly returns on an investment, and `time` (`1:25`) is the month number.

  We create a time-series plot, with tick marks every three months annotated as "Jan", "Apr", "Jul", and "Oct", by typing:

```
> plot(x = time, y = y, type = "l",          # Make the plot without an x-axis
+      xaxt = "n",
+      main = "Monthly Returns")

> axis(side = 1,                             # Add the x-axis
+      at = seq(from = 1, to = 25, by = 3),
+      labels = c("Jan", "Apr", "Jul", "Oct", "Jan", "Apr", "Jul", "Oct", "Jan"))
```



Section 10.3 Exercises

Exercise 9 Suppose two variables, x and y, were measured on males and females:

```
> x.m <- c(1, 3, 2, 6, 5, 5, 3, 4)          # Males
> y.m <- c(7, 7, 9, 6, 6, 8, 3, 5)


> x.f <- c(4, 4, 3, 7, 6, 8, 7, 9)          # Females
> y.f <- c(9, 11, 8, 8, 7, 8, 4, 5)
```

a) `points()` is useful for plotting different groups together in the same scatterplot using different plot characters or colors. After creating the vectors above, execute the following commands and report the result:

```
> plot(x.m, y.m, ylim = c(4, 12), xlim = c(1, 10),
       pch = 19, col = "blue")
> points(x.f, y.f, pch = 17, col = "red")
```

b) Now execute the following command and report the result:

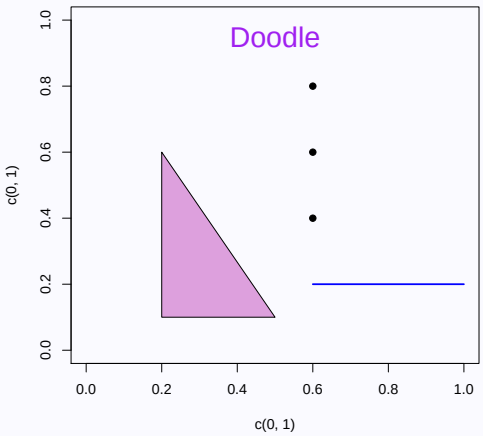
```
> legend(x = 8, y = 11, legend = c("Male", "Female"),
       pch = c(19, 17), col = c("blue", "red"))
```

Exercise 10 The following command sets up a blank coordinate system, to which we can add points, lines, polygons, text, etc. using `points()`, `lines()`, `polygon()`, `text()`, etc.

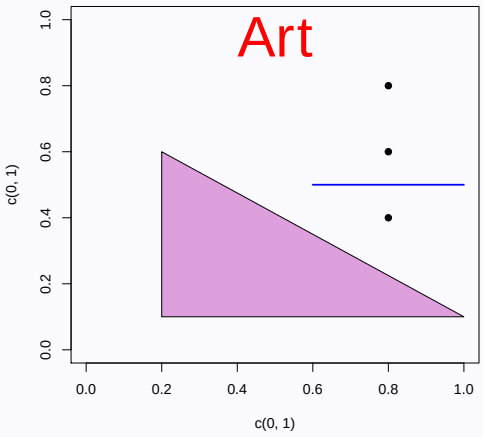
```
> plot(x = c(0, 1), y = c(0, 1),
      col = "transparent")          # Just set up the coordinate system.
```

Once the blank coordinate system is set up, the commands below produce the graphical display shown after them.

```
> polygon(x = c(0.2, 0.2, 0.5), y = c(0.1, 0.6, 0.1), col = "plum")
> points(x = c(0.2, 0.4, 0.6), y = c(0.8, 0.8, 0.8), pch = 19)
> lines(x = c(0.6, 1.0), y= c(0.2, 0.2), col = "blue", lwd = 2)
> text(x = 0.5, y = 0.95, labels = "Doodle", col = "purple", cex = 2)
```



After setting up the blank coordinate system, modify the commands above so that they produce the following graphical display. Report the changes that you made to the above commands:



10.4 Multiple-Figure Displays

10.4.1 Multiple-Figure Displays Using `mfrow` in `par()`

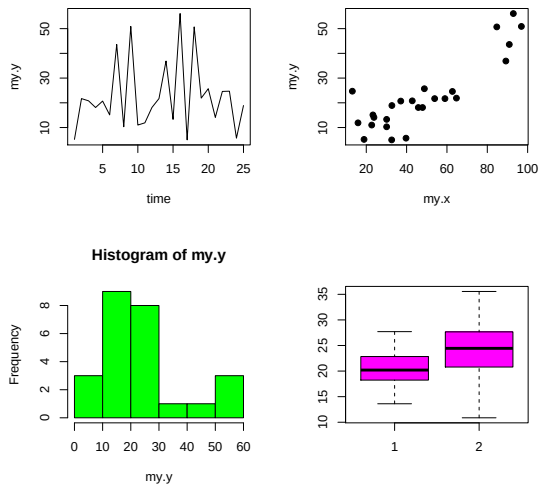
- One way to create a "multi-figure" array is by setting the `mfrow` graphical parameter, a vector of the form `c(nrow, ncol)`, using `par()`.
- For example, to make four plots in a two-row by two-column layout, first set `mfrow` by typing:

```
> par(mfrow = c(2, 2))
```

Then create the plots one by one:

```
> plot(x = time, y = my.y, type = "l")
> plot(x = my.x, y = my.y, pch = 19)
> hist(my.y, col = "green")
> boxplot(x1, x2, col = "magenta")
```

The result is:



Don't forget to reset `mfrow` back to its original setting, `c(1, 1)`:

```
> par(mfrow = c(1, 1))
```

10.4.2 Multiple-Figure Displays Using `layout()`

- Figures can be laid out in more complex arrangements using:

```
layout()        # Specify a complex figure arrangement
```

- `layout()` takes a matrix argument, `mat`, and divides the graphics window into as many rows and columns as there are in `mat`.

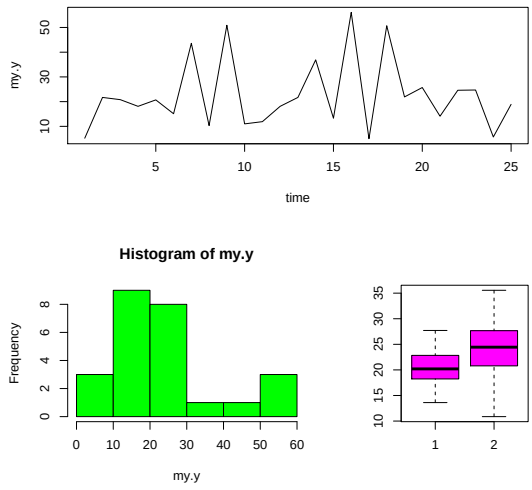
- Each value in `mat` should be either one of the integers $1, 2, ..., N$, where N is the number of figures to be included in the arrangement, or 0, and indicates which figure (or none if 0) should occupy the corresponding position in the layout.
- For example, the matrix:

```
> my.mat
```

```
     [,1] [,2]
[1,]    1    1
[2,]    2    3
```

indicates that we want $N = 3$ figures in the arrangement, the 1st of which should occupy the entire first row and the 2nd and 3rd of which should occupy their respective columns in the second row. Here's the result:

```
> layout(mat = my.mat, widths = c(1.5, 1))
> plot(x = time, y = my.y, type = "l")
> hist(my.y, col = "green")
> boxplot(x1, x2, col = "magenta")
```



The optional arguments `widths` and `heights` control the relative widths and heights of the columns and rows of the figure arrangement. Above, we specified that the first column should be 1.5 times as wide as the second.

10.5 Opening a New Graphics Window

- Each time we call `plot()` (or any other plotting function), the current graph will be replaced by the new one. If you don't want that to happen, use:

```
windows()      # Opens a new graphics device (on the Windows operating
               # system).  On Linux, use X11().  On Mac, use
               # macintosh().
```

- For example, if we want to compare two histograms of data vectors `x` and `y` side by side, we can type:

```
> hist(x)

> windows()          # X11() on Linux, macintosh() on Mac.

> hist(y)
```

10.6 Interactive Graphics Functions `identify()` and `locator()`

- These interactive graphics functions are used to identify points in a scatterplot and to locate coordinates in a plot:

```
identify()     # Used to identify points in a scatterplot by clicking
               # on them with the mouse
locator()      # Returns the x and y coordinates of a point in a plot
               # clicked on with the mouse
```

- Both involve clicking with the mouse in the graphics window.

10.6.1 Identifying Observations Using `identify()`

- As example of the use of `identify()` to identify a point in a scatterplot, after creating the plot:

```
> plot(x = my.x, y = my.y)
```

typing the following command and then left clicking a point in the scatterplot gives the index of the clicked point:

```
> identify(my.x, my.y, n = 1)          # Clicking on the 20th observation in
                                       # the graphics window identifies that
                                       # observation.

[1] 20
```

Thus the point that was clicked is `my.x[20]`, `my.y[20]`. (The argument `n` passed to `identify()` specifies how many points we want to identify.)

10.6.2 Determining Coordinates of Points in Plots Using `locator()`

- `locator()` enables us to determine the x and y coordinates of a point in a plot by clicking on its location. For example, typing:

```
> plot(x = my.x, y = my.y)

> clicked.coords <- locator(n = 1)
```

and then left clicking in the scatterplot gives the x and y coordinates of the clicked point:

```
> clicked.coords

$x
[1] 32.54422

$y
[1] 45.27469
```

(The argument `n` passed to `locator()` specifies how many points we want to locate.)

Section 10.6 Exercises

Exercise 11 Execute the following commands:

```
> xx <- c(6, 4, 7, 3, 9, 8, 11, 13)
> yy <- c(7, 6, 9, 5, 9, 11, 11, 15)
> plot(xx, yy)
```

a) Now try typing:

```
> identify(xx, yy, n = 1)
```

and then left clicking on a point in the scatterplot. What's the result?

b) Try typing:

```
> locator(n = 1)
```

and then left clicking anywhere in the graph. What's the result?

c) Try typing:

```
> text(locator(n = 1), labels = "Hello")
```

and then left clicking anywhere in the graph. What's the result?

10.7 Three-Dimensional Graphics and Surface Plots

10.7.1 Surface Plots

- Here are some useful functions for making three-dimensional plots and images of surfaces over two-dimensional domains:

```
persp()            # Graph a surface over the x,y plane.  You can use
                   # outer() to generate values z of the surface over a
                   # grid of x,y pairs.
image()            # Plot a grid of colored rectangles corresponding to
                   # the value of a surface over the x,y plane.
contour()          # Make a contour plot or add contour lines to an ex-
                   # isting plot.
filled.contour()   # Make a contour plot filled with colors
outer()            # Create a matrix z of values z = f(x, y), i.e.
                   # z[i, j] = f(x[i], y[j])
```

- For example, suppose we want to graph the bivariate function

$$f(x, y) = xy$$

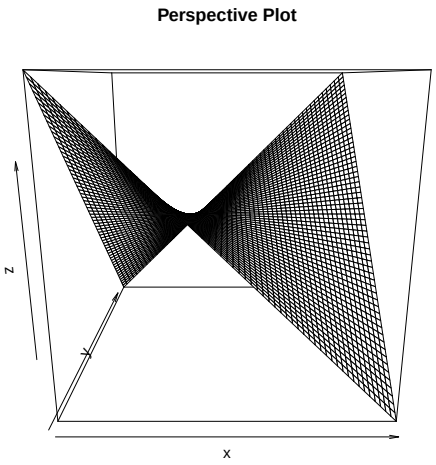
over the range $-3 \leq x \leq 3$ and $-3 \leq y \leq 3$.

First we need to set up x and y coordinates that define a two-dimensional grid in the x, y plane. Then we need to obtain values of $f(x, y)$ evaluated on those x, y grid points. The function `outer()` is used to evaluate a function $f(x, y)$ of two variables x and y over a two-dimensional grid in the x, y plane:

```
> x <- seq(from = -3, to = 3, by = 0.1)      # Create a sequence of x coordinates
> y <- seq(from = -3, to = 3, by = 0.1)      # Create a sequence of y coordinates
> z <- outer(X = x, Y = y, FUN = '*')        # Create a matrix z of values z = x*y
>                                            # (i.e. z[i, j] = x[i]*y[j])
```

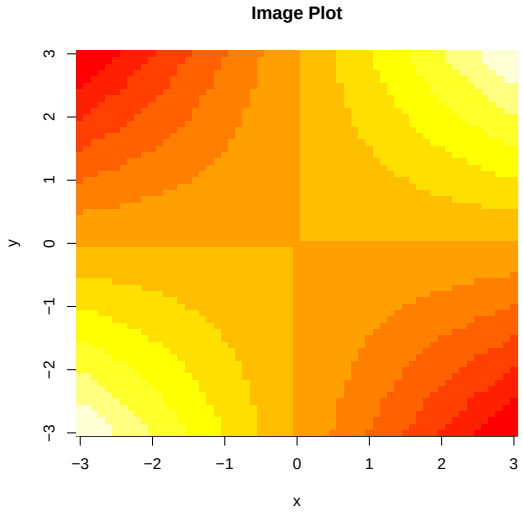
Now we're ready to make the plots. Here's the perspective plot:

```
> persp(x, y, z, main = "Perspective Plot")
```



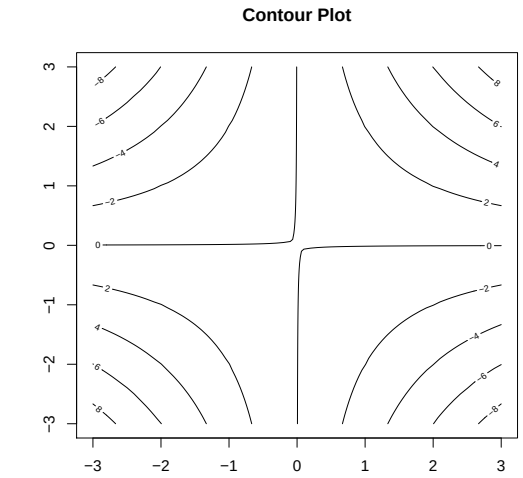
Here's the image plot:

```
> image(x, y, z, main = "Image Plot")
```



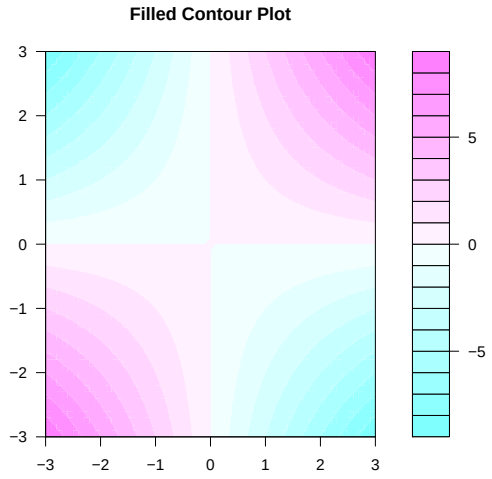
Here's the contour plot:

```
> contour(x, y, z, main = "Contour Plot")
```



Finally, here's the filled contour plot:

```
> filled.contour(x, y, z, main = "Filled Contour Plot")
```



10.7.2 Three-Dimensional Scatterplots

- The package `scatterplot3d` has a function `scatterplot3d()` that will produce three-dimensional scatterplots.
- To download and install the `scatterplot3d` package on Windows, from the Packages pulldown menu, select Install Package(s), choose a CRAN mirror nearby, then select `scatterplot3d` from the list.
- Once the package is installed, it can be loaded into the current R session by typing

  ```
  > library(scatterplot3d)
  ```

  Once this is done, the function `scatterplot3d()` will be available for use.