Bioinformatics and Computational Biosciences Branch
NIAID Office of Cyber Infrastructure and Computational Biology

NIH Intranet: http://bioinformatics.niaid.nih.gov
ScienceApps@niaid.nih.gov

Training Manual

Crash Course: Better Graphics in R

Jeff Skinner, M.S.

Bioinformatics and Computational Biosciences Branch (BCBB)
NIH/NIAID/OD/OSMO/OCICB
http://bioinformatics.niaid.nih.gov
ScienceApps@niaid.nih.gov

Table of Contents

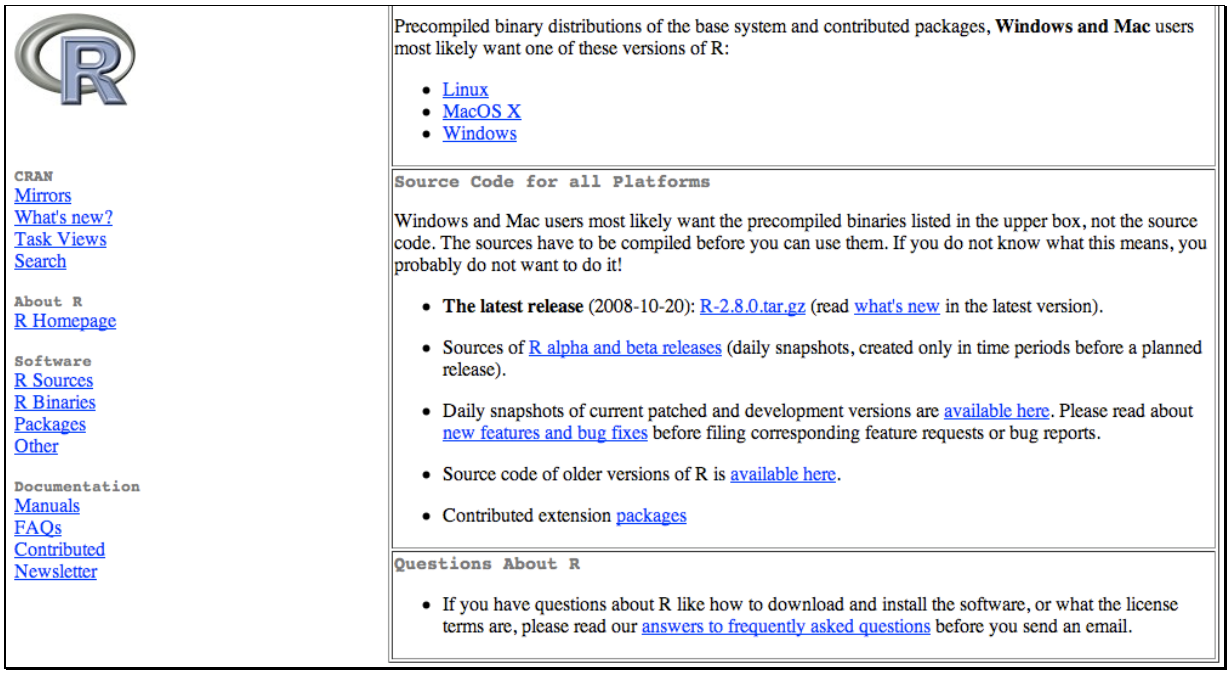
Ch. 1. Review of R Basics

1.1 What is R? What is BioConductor?

Many researchers know about the powerful analysis and visualization capabilities of R and BioConductor, even if they have not used R or BioConductor for themselves. R combines an open source software platform for statistics and data visualization with a powerful scripting language that is used to create new analyses and workflows. Both the software package itself and its scripting language are called R. While most people use R for statistical analyses and graphics, R can also be used for matrix algebra computations, data management and reporting. Advanced users will find that R interacts well with databases, commercial statistics software and several programming languages, so R can be utilized in many complicated computing solutions. BioConductor is an open source software development project that creates new tools for the analysis and comprehension of genomic-scale data. BioConductor include R package libraries for the annotation, normalization, filtering, statistical analysis and visualization of data from microarrays and other high-throughput biology experiments.

1.2 Download R

If you need to download R, visit the Comprehensive R Archive Network (CRAN) website (http://cran.r-project.org/) and look for the download links (Figure 1). Platform-specific download links for Linux, Mac OSX and Windows operating systems provide access to ready-to-use installations of R software. There are additional links to download individual source or binary files, so expert users can build their own custom installation of R. Remember that R is open source software, so everyone is welcome to modify its code and contribute to upcoming versions of the software. Most biology researchers should probably use the platform-specific download links, but remember that the custom installation options are available. Visit http://bioconductor.org to install Bioconductor.



Precompiled binary distributions of the base system and contributed packages, **Windows and Mac** users most likely want one of these versions of R:

- Linux
- MacOS X
- Windows

Source Code for all Platforms

Windows and Mac users most likely want the precompiled binaries listed in the upper box, not the source code. The sources have to be compiled before you can use them. If you do not know what this means, you probably do not want to do it!

- **The latest release** (2008-10-20): R-2.8.0.tar.gz (read what's new in the latest version).
- Sources of R alpha and beta releases (daily snapshots, created only in time periods before a planned release).
- Daily snapshots of current patched and development versions are available here. Please read about new features and bug fixes before filing corresponding feature requests or bug reports.
- Source code of older versions of R is available here.
- Contributed extension packages

Questions About R

- If you have questions about R like how to download and install the software, or what the license terms are, please read our answers to frequently asked questions before you send an email.

Figure 1. The Comprehensive R Archive Network (CRAN) website.

1.3 Helpful Resources

Because R is a open source software program, there is no corporate office to call or email for technical support. However, there are many resources available to help users learn how to use R or answer technical questions about R. Visit the R project website (http://www.R-project.org) to find free manuals, a FAQ page, a list of published books on R, the R Wiki and various mailing lists. You can find documentation of specific R functions by using its `help()` commands, as demonstrated in section 1.5 of this manual. Some historic books on the S and R software packages include “the blue book” (Becker et al. 1988), “the white book” (Chambers and Hastie 1992) and “the green book” (Chambers 1998), but there are now dozens of statistics and programming text books for the R and S languages. The R-help mailing list is sometimes your best bet for person-to-person help with R and its functions, but it is important to read the posting guide before posting new messages to the mailing list.

1.4 R GUI for Windows

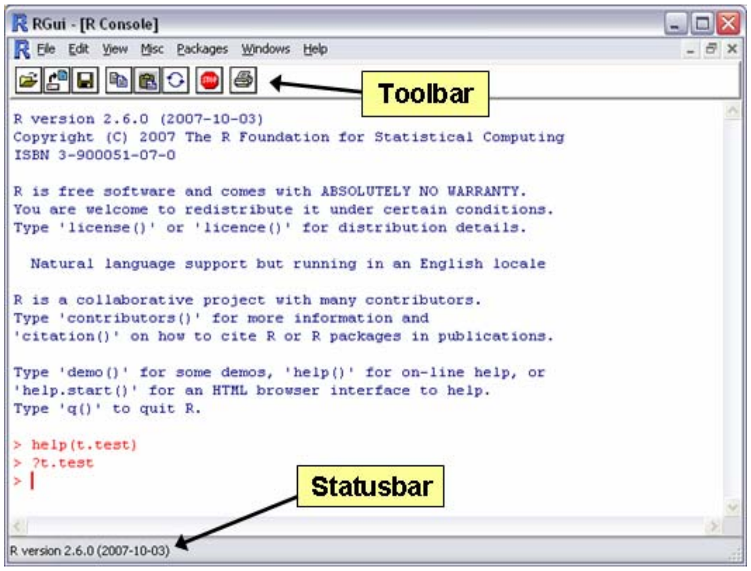
Although some programmers and statisticians chose to use R from the command line of a computer terminal, most biologists and researchers will use R from Windows or Mac graphic user interface (GUI). The current R GUI in MS Windows features seven clickable menus and eight clickable buttons to help you access commonly used features (Figure 2). The **File** menu allows you to create, open or run source scripts; load or save R workspace environments; or change your working directory. File menu options *Open script…*, *Load workspace…*, *Save workspace…* and *Print* are also available on buttons of the Windows R GUI Tooldbar (Figure 2). The **Edit** menu includes *Copy* and *Paste* options, plus the *GUI preferences* window. *Copy* and *Paste* are also available on the Toolbar. The **View** menu can hide or display the *Toolbar* and the *Statusbar* (Figure 2). The **Misc** includes options to control the R workspace. The **Packages** menu includes options to browse, download and install package libraries from know repositories. The **Windows** menu allows you to switch between open windows within the GUI, while the **Help** menu provides several options to search for help on R features and commands.

1.5 Searching the Help Menus

There are many books, guides and manuals available to help you learn how to use R, but inevitably every R user must search the help menus. The R help menus can help you find new functions or provide more detailed explanations of the inputs and outputs of a function you have already used. There are several useful help commands in R to find the documentation that you need.

1.5.1 Help documentation with `help()` and `?`

The two function `help()` is used to find documentation for a specific function, when you already know the command name. For example, the command `help(t.test)` is used to find the documentation for the `t.test` command, which is used to compute the Student’s t-test (Figure 2). Try entering `help(log)` or `?log` for another example. These help commands are most useful if you need a detailed explanation of a function you already use or if you would like to investigate a function you found in a paper or on the internet. The two commands are equivalent.


RGui - [R Console]
File Edit View Misc Packages Windows Help
Toolbar
Statusbar
R version 2.6.0 (2007-10-03)

```
R version 2.6.0 (2007-10-03)
Copyright (C) 2007 The R Foundation for Statistical Computing
ISBN 3-900051-07-0

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> help(t.test)
> ?t.test
> 
```

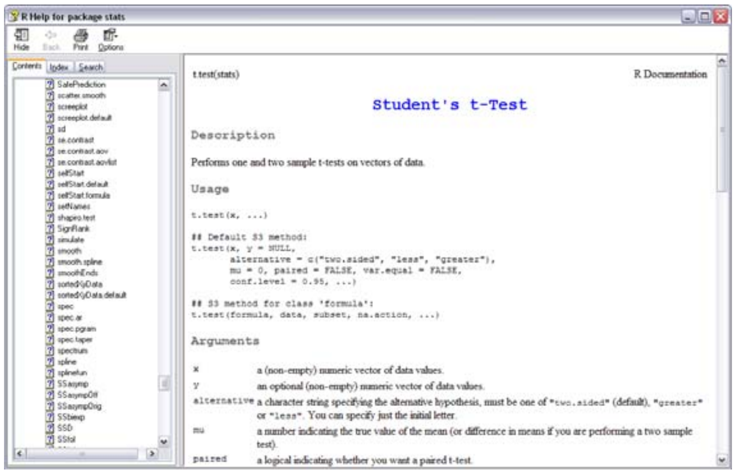
Figure 2. The MS Windows R GUI with Toolbar and Statusbar features identified.

From this point forward, all commands described in this manual will be shown in boxed and indented text, as shown below:

```
> help(t.test)   # Find documentation for the function t.test
> ?t.test        # (same as above)
>
```

Notice that all text appearing after the # symbol will be ignored. The # symbol is a restricted character in R that is used to add comments to the R scripting language. These comments can be a very useful feature, particularly if you need to share code with others.

Both the `help()` and `?` commands produce an HTML-formatted manual for the specified function (Figure 3). You will notice that most of the help documentation for R functions follows very strict formatting guidelines. Each documentation page provides the command information to call the function, the function's name, a description of the function purpose, details about its usage within R, details about its arguments, details about its output and typically a specific example that you can copy-and-paste into R for demonstration purposes.



t.test(stats) R Documentation

Student's t-Test

Description

Performs one and two sample t-tests on vectors of data.

Usage

```
t.test(x, ...)

## Default S3 method:
t.test(x, y = NULL,
       alternative = c("two.sided", "less", "greater"),
       mu = 0, paired = FALSE, var.equal = FALSE,
       conf.level = 0.95, ...)

## S3 method for class 'formula':
t.test(formula, data, subset, na.action, ...)
```

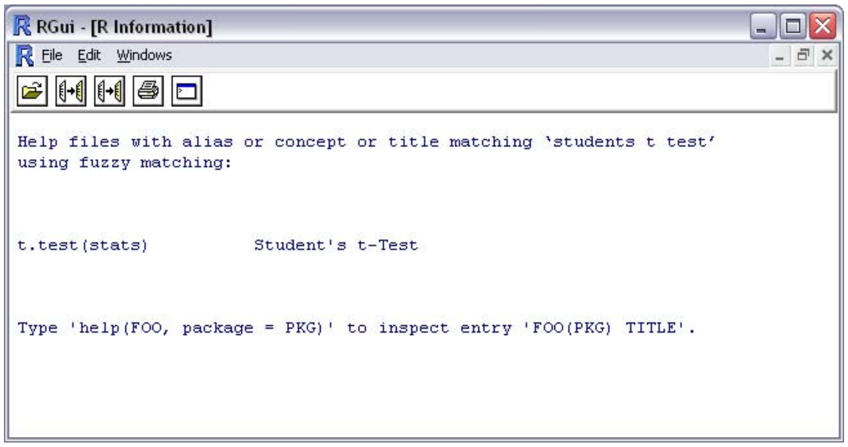
Arguments

x	a (non-empty) numeric vector of data values.
y	an optional (non-empty) numeric vector of data values.
alternative	a character string specifying the alternative hypothesis, must be one of "two.sided" (default), "greater" or "less". You can specify just the initial letter.
mu	a number indicating the true value of the mean (or difference in means if you are performing a two sample test).
paired	a logical indicating whether you want a paired t-test.

Figure 3. Help documentation for the function `t.test`.

1.5.2 Keyword searches with `help.search()`

Another type of search is required when you do not know the command of a specific function. Type `help.search("keyword")` to search for keywords and find all the command names of functions related to your keyword. For example, the command `help.search("students t test")` will produce a list of all functions related to the student's t-test (Figure 4). In this example, only the `t.test()` function is found by our search, but other requests may generate many results.



```
Help files with alias or concept or title matching 'students t test'
using fuzzy matching:



t.test(stats)          Student's t-Test



Type 'help(FOO, package = PKG)' to inspect entry 'FOO(PKG) TITLE'.
```

Figure 4. List of help files from *help.search* keyword search.

1.5.3 Google and other search engines

One final suggestion is to use Google.com or other search engines to find help with R. I recommend that you always include the keyword "CRAN" or "BioConductor" in your R-related Google searches, because it can help direct you to search results that are most directly related to R packages and concepts (Figure 14). The search engine Rseek (http://www.rseek.org/) is a search engine that only queries the R help files and related websites.

1.6 Installing R Packages and Source Scripts

Help searches and basic arithmetic functions are included in the base R software package, but often researchers need to use specialized research tools that are not included in the base R software. These specialized tools are often available as free downloadable packages or source scripts in R. Packages are user-submitted R scripts and functions that have are made been posted online by CRAN, BioConductor or other websites. Packages are downloaded and installed from the R GUI or the command line. The code for these packages is typically downloaded as a source file, written in the R scripting language, or as a binary, written in a compiled language like C or FORTRAN. Some functions and scripts have not been submitted as packages to CRAN or BioConductor, but they still may be loaded into your installation of R as a source file.

1.6.1 R packages

Click > **Packages** > **Install Package(s)…** to install packages from the Windows R GUI. If you are installing packages for the first time, you may be prompted to *Set CRAN mirror…* or *Select Repositories…*before you continue (Figure 5). Remember, the CRAN mirror site is a server that contains the most recent R software downloads and packages. You want to choose one of the CRAN mirrors nearest you for convenience. The package repositories are specific lists of packages from CRAN and BioConductor. Choose only the repositories you need, because selecting more repositories will create a longer list of packages for you to browse.

Use the scroll bar to browse through the list of R and BioConductor packages and select the packages you need for installation (Figure 16). You can hold the Ctrl key to select multiple packages, if necessary. Note the list of packages can be very long, especially if several repositories were selected. Once you have selected the R packages you need, click OK to download and install the packages.

Alternatively, if you know the name and repository address of the packages you need to download, you can download and install a package from the command line using the command `install.packages()`. There is no advantage or disadvantage to using the R GUI or the command line to install a package, but the `install.packages()` command can be very helpful when scripting. Using the `install.packages()` command in your source code will ensure any users of your script will have all the necessary R packages. As the packages download, you may see some log messages in your R console to keep you informed of the download progress and any potential errors. After the packages have finished downloading, you will want to enter a `library()` or `require()` command for the package to load the contents of the package into your R workspace. Both commands have the same function, but the `require()` is preferred for use within R functions.

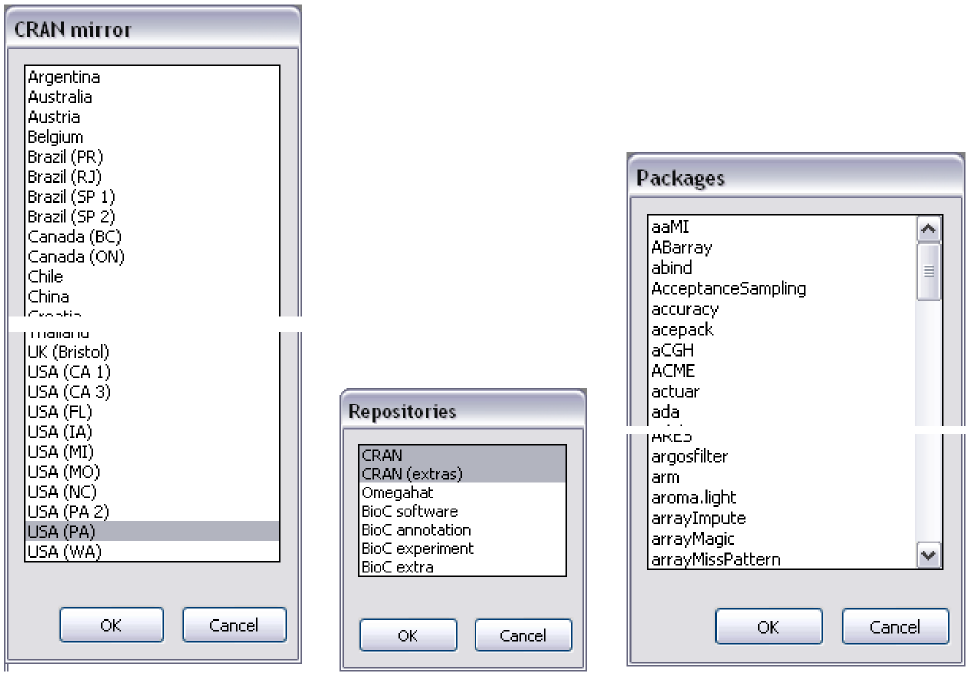


Figure 5. *CRAN mirror*, *Select Repositories* and *Packages* menus in Windows R GUI.

```
> install.packages("gtools",repos="http://cran.r-project.org")
trying URL 'http://cran.r-project.org/bin/windows/contrib/2.6/gtools_2.4.0.zip'
Content type 'application/zip' length 157621 bytes (153 Kb)
opened URL
downloaded 153 Kb

package 'gtools' successfully unpacked and MD5 sums checked

The downloaded packages are in
        C:\Documents and Settings\skinnerj\Local
Settings\Temp\RtmpIQ1HTc\downloaded_packages
updating HTML package descriptions
> library(gtools)
```

1.6.2 Source scripts

The R command language does not need to be entered one line at a time on the command line; it is also possible to store multiple R commands in a single R script file (extension *.R*) or in a simple text file (extension *.txt*) that can be run from the command line. These R scripts can be used to string multiple commands together, automating an otherwise lengthy workflow, or they can be used to define entirely new functions. Often users will choose to distribute these R scripts rather than creating an R package library, so keep in mind that you can also upload new functions using R source scripts. If you have found a R source script that you need to upload, click > **File** > **Source R code…** on the Windows R GUI. Use the command `source()`, if you prefer to load the source script from the command line.

```
> # Load a source script file (.R extension)
> source("~/example.R")
```

1.7 Entering and Importing Data

There are dozens of ways to enter data into R. Many famous and historical datasets are already uploaded and available for use in the base R software or in an R package. There are also hundreds of functions that make it easy to type and enter small data sets manually into R. Other functions can help to generate huge amounts of random or simulated data. Finally, there are a variety of functions available to help you upload your own data files, whether they are stored as R workspace data (*.Rdata*), as plain text files (*.txt*, *.csv*, …), in proprietary data file formats from other popular statistics packages (*.sav*, *.sas7DAT*, …), as MS Excel spreadsheets (*.xls*, *.xlsx*, …) or even tables from a database (e.g. MS Access, MySQL, …).

1.7.1 Base and package data in R

Enter the command `data()` or click > **Packages & Data** > **Data Manager** in the Mac OSX R GUI to view a list of the datasets currently available on your installation of R (Figure 17). Try to find the data set “iris” in the list. Select the “iris” data set from the list, or enter the help command `?iris`, to read the documentation describing this data set. The “iris” data set in R is the famous “Fisher’s iris data”, originally collected by biologist Edgar Anderson. Enter the command `iris` to view the Fisher’s iris data set, already stored in R. Additional data from textbooks and from published biological experiments (e.g. microarray, ...) can be downloaded as specific R package libraries.

1.7.2 Entering data manually

The base and package data available in R can be a useful resource, but most R users need to upload their own data into R. Most researchers will already have their data stored in a large data file. However, for some small data sets, it may be easiest to enter the data manually from the command line. In other situations, researchers may need to simulate large amounts of data using procedures from the command line. Some common R procedures are used to generate a small dataset concerning the Alpha-fetoprotein (AFP) levels of 20 medical patients in the source files *afpImport.R* or *afpScript.R*, available at our website (http://collab.niaid.nih.gov/sites/research/SIG/Bioinformatics/seminars.aspx).

The source script uses the command `subject <- 1:20` to generate the sequence of integers from 1 to 20 for use as subject IDs, stored as a variable named *subject*. The command `males <- rep("male",10)` is used to generate a vector of 10 replicated string values to label the male patients. A similar replicated vector is created to identify 10 female patients, then the combine command `c()` is used concatenate the two vectors together and store a variable named *gender*. The command `rnorm(10,70,2.5)` is used to randomly generate 10 new observations from a Gaussian normal distribution with mean 70 and standard deviation 2.5 to represent the heights of our male patients, while `rnorm(10,64,2.2)` will generate the height data for our female patients. The commands `runif(10,155,320)` and `runif(10,95,210)` generate random weight data from a random uniform distribution for the male and female data, respectively. The formula BMI = (703 * mass (lbs)) / (height^2) is used to compute BMI estimates. Finally, the `seq()` and `rep()` commands are used to generate a set of repeated doses for the patients. All the new variable columns are joined together using the `data.frame()` command to finish, then view the results by entering the data set name `afp.data` at the command line.

1.7.3 Importing previously saved R data workspaces (*.RData*)

Not surprisingly, it is possible to save and load R data sets in their own file format (file extension *.RData*). Use the `load()` command to import a previously saved *.RData* file. The `load()` command only includes two parameters, the `file` parameter used to specify the filepath of the *.RData* file that will be imported and the more complicated `envir` parameter that specifies an `environment` for the uploaded R workspace. Briefly, an R `environment` is a collection of named objects in R. The user's R session workspace is an `environment`, for example. If during one session of use, an R user defines the variable `aa = 42.7`, then any reference to the variable `aa` will return the value 42.7 until the variable `aa` is redefined or until the workspace is closed. If you load or import a saved R data workspace, then all the variables and objects defined in that workspace will be retained.

1.7.4 Text file data (*.txt*, *.csv*) with `read.table()`

The most efficient way to import data into R is to upload a text file. Text files are typically smaller than proprietary data formats, like MS Excel spreadsheets. Since plain text files are not organized by columns, rows and cells like a MS Excel spreadsheet, users will need to specify a character to separate the values of different fields (i.e. a *delimiter* to separate columns) and a character to mark the end of each line of text (i.e. the end of a row). Often, the first row of a text data file is used to name the fields (i.e. columns) of a data table. Sometimes strings of character data are enclosed with single or double quotation marks to avoid conflicts with delimiter symbols and numeric fields. These issues are addressed in the parameters of the text file import procedures.

The most popular way to import a plain text data file is with the `read.table()` command. The `read.table()` command is the most general method to read table style data from a plain text data file. Suppose you have two different data sets. The first data set is a tab-delimited text file named *AlphaFetoProtein.txt* that contains the AFP data described in section 1.7.2 above. The second data set is a comma separated value (*.csv*) plain text file named *AdverseEvents.csv* that contains epidemiological data on the demographics and adverse event symptoms of 64 medical patients. Both files can be opened with the `read.table()` command.

```
> # Import a tab-delimited text for data set named AFP
> #
> AFP <- read.table(file = “~/AlphaFetoProtein.txt,header = TRUE, sep = “\t”,
                            stringsAsFactors = FALSE)
> AFP
> #
> # Import a comma separated value text file data set named ‘AE’
> #
> AE <- read.table(file = "~/AdverseEvents.csv", header = TRUE,
                   sep = ",", stringsAsFactors = TRUE)
> AE
```

The `read.table()` procedure include the parameter `file` to specify the quoted file path of the data file that will be imported. The `header = TRUE` parameter indicates that the data file has a header line to define the column (i.e. variable) names of the data table. If the header is not specified, column names can be entered using the parameter `col.names`. The parameter `sep = "\t"` indicates the AFP data file should be imported as a tab-delimited text files (i.e. columns are separated by tabs). Likewise, the parameter `sep = ","` indicates that fields are separated by commas in the AE data. The parameter `stringsAsFactors = FALSE` indicates that all string variables in the data file will be stored as character data rather than factor data. Many statistical and graphing procedures require character data to be specified as factors, but it may be easier to modify data tables if the string variables are stored as character data. The `read.table` command includes many other parameters that could be useful. E.g. the parameter `dec = ","` specifies that numbers are recorded with European-style "comma" decimals (e.g. 2.63 = 2,63).

The `read.csv()` and `read.csv2()` commands are equivalent to `read.table()` with default parameters optimized for comma separated value text files. Specifically, the `read.csv()` command is optimized for 'American' formatted .csv files, where fields are separated by commas and decimals are separated from integers with a period. 'European' formatted .csv files, where fields are separated by semicolons and decimals are separated from integers with a comma, should be imported with `read.csv2()`. Similarly, the `read.delim()` and `read.delim2()` commands are equivalent to `read.table()` with default parameters optimized for importing tab-delimited text files. Specifically, the `read.delim()` command is optimized for 'American' formatted .txt files, where fields are separated by tabs and decimals are separated from integers with a period. 'European' formatted .txt files, where fields are separated by tabs and decimals are separated from integers with a comma, should be imported with the `read.delim2()` command.

1.7.5 MS Excel files and other proprietary formats

Historically, it was difficult to import data from MS Excel spreadsheets into R. Most people will convert their MS Excel spreadsheets into tab-delimited text files (*.txt*) or comma-separated value text files (*.csv*), then import these text files into R using the `read.table()` or `read.csv()` commands. However, converting Excel spreadsheets into text files may be tiresome, if dozens of files need to be converted, and some users may not have access to MS Excel to convert *.xls* spreadsheets into .txt files. It is now possible to import MS Excel spreadsheets directly using the `read.xls()` command from the `xlsReadWrite` package library.

Click > **Packages** > **Install package(s)…** on the MS Windows R GUI or click > **Packages & Data** > **Package Installer** on the Mac OSX R GUI to find and install the `xlsReadWrite` package library. Enter the command `library(xls.ReadWrite)` to load the package library to your workspace. Upload your MS Excel data using the command:

```
bb <- read.xls(file="C:\\ sample.xls",colNames=TRUE,sheet=1)
```

Note, the statement `bb <- read.xls()` implies that we are defining a data set named bb, just like the previous example. Similarly, the parameter statements `file = "H:\\BCBB tips\\sample.xls"`, `colNames = TRUE` and `sheet = 1` indicate the file path of the MS Excel spreadsheet, choice to read column names from the first row of data and the choice to only read data from the first sheet of the MS Excel file, respectively.

Other R package libraries are available to open data files created or saved using commercial statistics software packages like SAS or SPSS. For example, SAS datafiles and SAS XPORT format libraries can be imported with the commands `read.ssd()` and `read.xport()` from the `foreign` package library. It is also possible to import SAS data sets and SAS Transport files using the `sas.get()` and `sasxport.get()`

commands from the `Hmisc` package library. Similarly, the functions `spss.get()` and `read.spss()` from the package libraries `Hmisc` and `foreign`, respectively, can both be used to open SPSS data files (*.sav* file extension) in R. The command `stata.get()` from the `Hmisc` package library can be used to import Stata datasets into R, etc. Data sets from most commercial statistics software packages can be imported directly.

1.8 Basic Manipulation of Data

There are a many ways to manipulate data sets in R, but some methods will be crucial to the rest of this manual. Specifically, users will need to be able to call specific columns, rows and variables from the data tables they have stored in R using the import methods described above.

1.8.1 Indexing

One of the most important concepts in R is the idea of indexing, because it applies to so many types of R objects. Vectors, matrices, data frames, arrays and lists can all be indexed using similar command notations. The index of an R object refers to the specific location of a value in a vector, matrix, array, data frame or list. You can generalize this concept by thinking of the index as the row and column number of any value entry in a spreadsheet, but remember that some R objects can have more than two dimensions or fewer than two dimensions. Here are some examples:

```
> # Report the third entry from a vector of length = 6
> Vector[3]
[1] 53
> # Report the entry from the 2nd row and 5th column of a matrix
> Matrix[2,5]
[1] 49
> # Report the 3rd row, 2nd column and 2nd panel of an array
> Array[3,2,2]
[1] 49
> # Report the 3rd row, 2nd column and 'Female' gender of a table
> Table[3,2,"Female"]
[1] 4
> # Report the 1st entry of the 1st column from AFP
> AFP[1,1]
[1] 1
```

Generally, you refer to an indexed entry of an R object by adding square brackets after the objects name (e.g. `Vector[3]` refers to the 3rd entry of the object `Vector`). The dimensions of an object are separated by commas (e.g. `Matrix[2,5]` refers to the 2nd row and 5th column of the object `Matrix`). If the dimensions of an object are named instead of numbered, then those dimensions can be specified with a quoted character string (e.g. specify the `"Female"` of the `gender` dimension). The examples above use indexing to report single values from vectors, matrices, arrays, tables, data frames and lists, but an index can be used in more complicated ways.

```
> # Rows 2-3 and columns 1, 2 and 6 of a matrix
> Matrix[2:3,c(1,2,6)]
     [,1] [,2] [,3]
[1,]   49   45   46
[2,]   45   56   51
```

```
> # Overwrite one value from a matrix
> Matrix[3,3] <- NA> Matrix
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]   49   54   49   47   57   53
[2,]   49   45   50   50   49   46
[3,]   45   56   NA   52   46   51
[4,]   47   46   48   48   55   48
> # Identify observations with % Body fat less than 10%
> AE[Percent.Body.Fat <= 10,]
      Region Gender Severity Age   Weight Percent.Body.Fat
2  Southwest   Male     Mild  34 148.5672                7
30 Southwest   Male     Mild  36 155.3823                8
49   Midwest   Male Moderate  34 151.3767                9
```

A sequence of row or column numbers can be entered into an index to view more than one row or column from a data table. These sequences can be entered using colon symbol notation (e.g. `1:5`) or the combine function (e.g. `c(1,4,7)`) and other methods. The individual indexed values of a matrix, array or data frame can be overwritten without affecting any other values in the matrix, array or data frame. Sequences of row numbers or column numbers can be generated with an inequality or conditional statement to find special subsets of data (e.g. all patients with Percent.Body.Fat <= 10%). Indexing is a very powerful tool within R. Indexing will be using in many of these examples.

1.8.2 Column references and attach()

Indexing can be a great way to create, view or modify subsets of your data, but often it might be more helpful to refer to specific columns, or variables, within a large data frame. We can also call a specific column of a data frame using the reserved dollar sign symbol (e.g. `AE$Gender` yields the gender column of the AE data set). Column references can be simplified using the `attach()` command to "attach" a specific data frame or list to the current R workspace. Once the object has been attached to the workspace, individual variables or list items can be called by name (e.g. After using `attach()`, `Gender` will be a new variable in your workspace).

Ch. 2. Simple Graphics and Customizations

2.1 Basic Types of Graphics and Figures

You can use R to produce dozens or hundreds of different kinds of graphics and figures. Many popular types of graphs, like pie charts and histograms, have their own dedicated commands and procedures in the `graphics` package library. Other types of graphs, like multifactor XY scatterplots, are most easily produced using multiple commands from general graphing utilities, like `plot()` and `legend()`. Often, specialized package libraries will include graphics commands that can help streamline the graphing process. Other graphs can only be produced in the context of the appropriate statistical analysis. Several simple examples are provided below.

2.1.1 Pie charts

Pie charts are used to quickly display the frequencies of each outcome of a single categorical variable. The relative size of each slice of the pie chart represents the relative frequency of its respective outcome in the sample. For example, we could use a pie chart to examine the proportion of samples from each iris species in

Edgar Anderson's iris data (Figure 6). We could also use a pie chart to explore the frequencies of each adverse event in our AE data set (Figure 7).

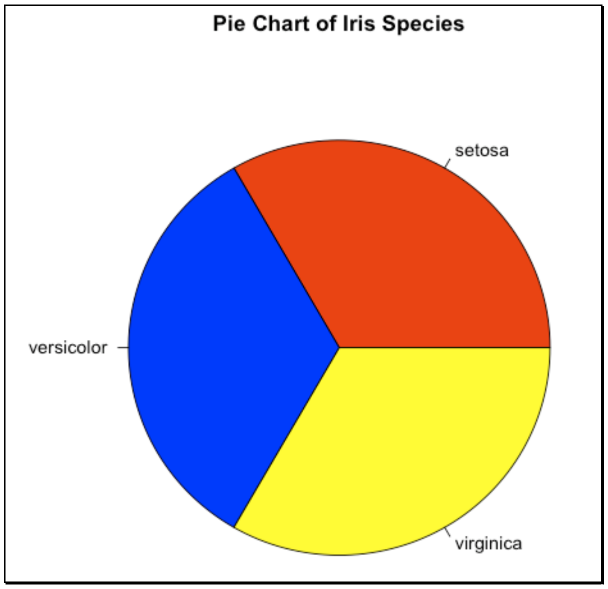


Figure 6. A pie chart of Edgar Anderson's iris species counts

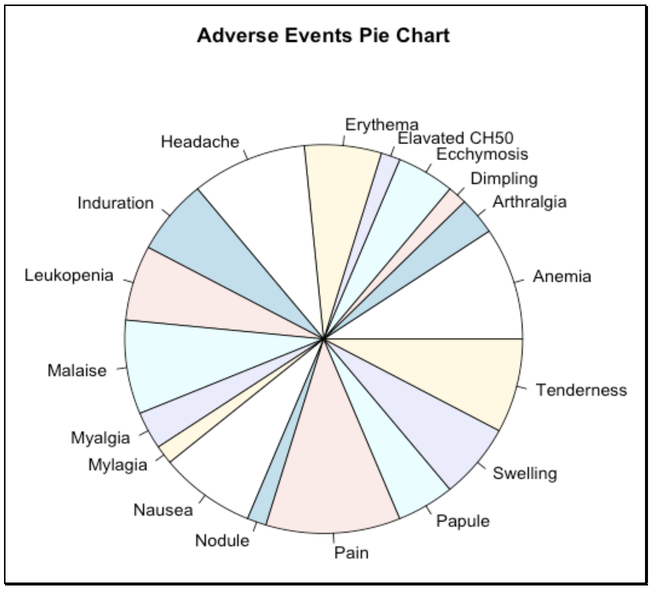


Figure 7. A pie chart of the adverse events (AE) data set.

```
># Create the labels for the iris data pie chart
> labels <- levels(iris$Species)
># Create a vector with all three species counts
> counts <- summary(iris$Species)
># Define a vector with three color choices for the pie chart
> colors <- c("red","blue","yellow")
># Define a main title for the pie chart
> main   <- "Pie Chart of Iris Species"
># Call the pie() command to produce the pie chart
> pie(x = counts, labels = labels, col = colors, main = main)
>#
># Create the labels for the adverse events (AE) data pie chart
> labels <- levels(as.factor(AE$Adverse.Event))
># Create a vector with counts for all adverse events
> counts <- summary(as.factor(AE$Adverse.Event))
># Define a main title for the pie chart
> main   <- "Adverse Events Pie Chart"
># Call the pie() command to produce the pie chart
> pie(x = counts, labels = labels, main = main)
```

The `pie()` command includes the parameter `x` to define the counts or frequencies in each slice of the pie chart, the parameter `labels` to define the text labels in each slice of the pie chart and the parameter `col` to define the colors of each slice in the pie chart. You can also add generic graphing parameters, like `main` and others, to customize the pie chart with a main title and other features. Notice that the `labels` and the `x` (counts or frequencies) parameters could have been computed and entered manually, but instead the commands `levels()` and `summary()` were used to define `labels` and `x`, respectively. The `levels()` command lists all the outcomes of a factor variable, while the `summary()` command adds up all the counts for each outcome of a factor variable. Note, the species variable of the iris data set was already defined as a factor variable, while the adverse events variable from the AE data set needed to be converted to a factor variable first, using the `as.factor()` command. Also notice, in the second pie chart, that the `col` parameter was left undefined and R automatically generated the color choices for each of the 18 adverse event slices.

2.1.2 Histograms

Histograms are used to quickly display the distribution of a single continuous numeric variable. Often researchers want to determine if a variable might be normally distributed or non-normally distributed. Other researchers want to estimate descriptive statistics like means, medians, variances or ranges. A key issue in the construction of a histogram is the choice of the histogram "bins" or groupings. If too many bins are used, the true shape of the distribution will be lost because the histogram will be too sparse, but if too few bins are used, the true shape of the distribution will be lost because the bins are too dense to remain informative. The location of the bin mid points and break points can also be important to the shape of the histogram. The importance of binning is shown in two histograms of the height measurements from the AFP dataset shown below (Figure 8 and Figure 9).

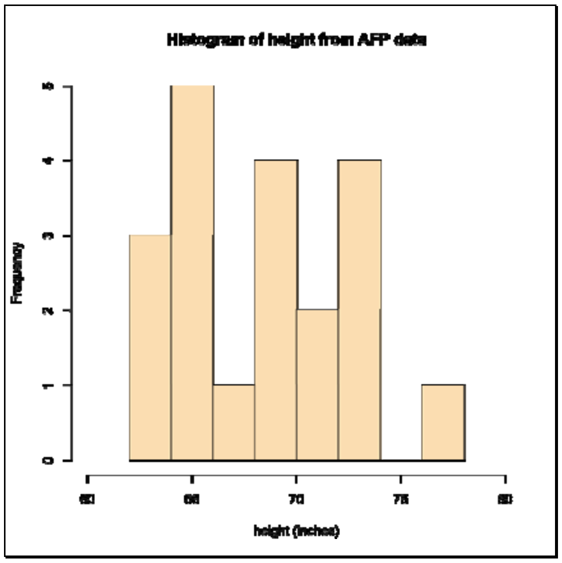


Figure 8. A histogram of height from the AFP data set with default number of bins.

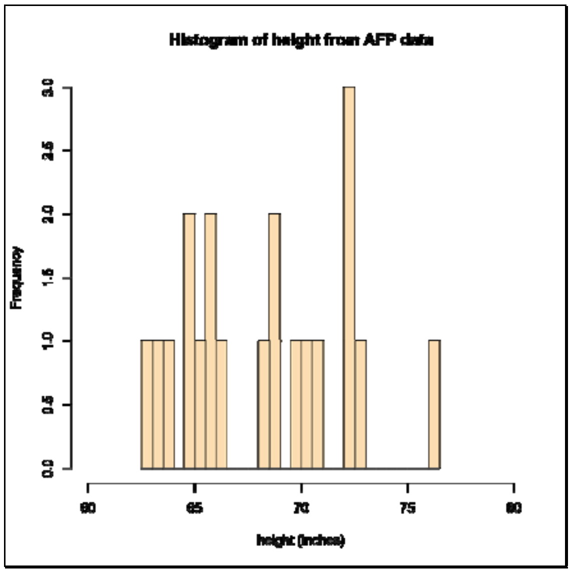


Figure 9. A histogram of BMI from the AFP data set with a larger number of bins.

```
> # Define a vector of BMI data
> height <- as.numeric(afp.data[,3])
> # Define a main title for the histogram
> main <- "Histogram of height from AFP data"
> # Call the hist() command to produce the histogram
> hist(x=height,xlab="height (inches)",main=main,col="wheat")
> # Call hist() command with extra breaks for a second histogram
> hist(x=height,breaks=30,xlim=c(15,45),...)
```

The `hist()` command includes many parameter options. The parameter `x` must be specified, to identify the sample of continuous data displayed in the histogram. The `breaks` parameter specifies the number of bins used in the histogram. The number of histogram bins can be specified using one of three automated binning algorithm choices (i.e. "Sturges", "Scott" or "Freedman-Diaconis"), a single number (i.e. breaks = 30 will produce 31 bins), a vector of specific break points or a formula. In the first histogram, the default "Sturges" method produced a histogram with six bins, which appears to show a normal distribution. In the second histogram, the command breaks = 30 specified that 31 bins should be used, and the resulting histogram was sparse and uninformative. The command `xlab` specifies the label for the x-axis. As before, the commands `main` and `col` specify the main title and the color of the plotted bars, respectively.

2.1.3 Box plots

Box plots are an alternative to the histogram for researchers who want to quickly summarize the distribution of continuous numeric variables. Box plots were introduced by statistician John Tukey in his historic book Exploratory Data Analysis (Tukey 1977). The box plot is a graphical representation of the five number summary, where the central line in the box plot represents the median of a sample, the outer edges of the box in the box plot represent the 25th and 75th percentiles of the sample and the whiskers of the box plot represent the minimum and maximum of a sample. Alternate versions of the box plot often use dots or asterisks to identify outliers beyond the whiskers, which might represent the 5th and 95th percentile of a distribution or the smallest and largest "non-outlier" values of a distribution. Generally, a single box plot (Figure 10) provides less information about the shape of a distribution than an analogous histogram. For example, a box plot cannot be used to identify a bimodal distribution of female and male heights, while a histogram can. However, box plots are often more appropriate than histograms when researchers want to compare the distributions of several samples in the same figure (Figure 11).

```
> # Define a vector of height data
> height <- as.numeric(afp.data[,3])
> # Define a main title for the boxplot
> main <- "Boxplot of height from AFP data"
> # Call boxplot() command for boxplot of height from AFP data
> boxplot(x=height,main=main,xlab=”height (inches)”,col=”wheat”)
> #
> # Call boxplot() command for boxplot of calories from AE data
> boxplot(formula=Calories~Region,data=AE,range=1.5,...)
```

The `boxplot()` function can be used in at least two different ways, with a single vector of continuous data or with a formula to produce side-by-side box plots. A simple box plot of the height variable from the AFP data set is produced using the `boxplot()` command with parameter `x = height` to specify a single vector of numeric data for the `Calories ~ Region` was used to create a graph with side-by-side box plots of the calories variable for each of the five regions of the categorical region variable. The parameter `data = AE` is used to specify that we only want to use variables from the AE data set, which is why we could call the calories and region variables with out defining them as vectors before the `boxplot()` command. The `range` parameter is

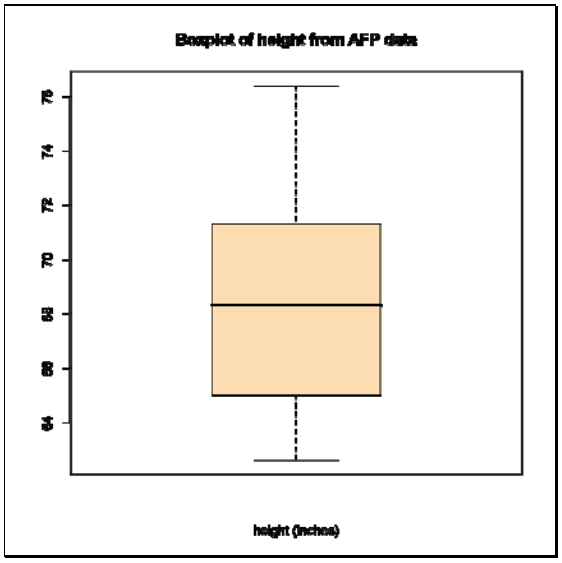


Figure 10. A box plot of patient height from the AFP data

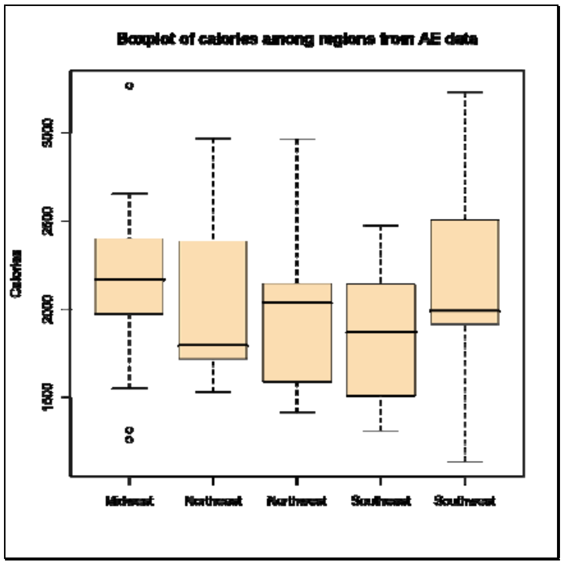


Figure 11. A box plot of calories intake among five regions from the AE data.

used to identify outliers in the box plot. The parameters `main`, `xlab` and `col` were used to specify the main title, x-axis label and the color of the boxplot, respectively, as seen in the previous examples. In the second box plot, the parameter `formula =` on the box plot figure. Here, `range = 1.5` implies that any calorie measurement smaller than Q1 – 1.5*IQR and any measurement larger than Q3 + 1.5*IQR will be identified as an outlier, where Q1 represents the 25th percentile, Q3 represents the 75th percentile and IQR represents the interquartile range (i.e. Q3 – Q1, the middle 50% of the data). Another parameter `ylab = "Calories"` was entered to specify the y-axis label, while the `main` and `col` were used to define a main title and the box plot color as above.

2.1.4 Simple bar charts

Researchers and statisticians often use the phrase "bar chart" to describe two subtly different types of graphs. Sometimes a bar chart is used as an alternative to the pie chart to display the relative frequencies of different outcomes from a categorical variable (e.g. gender or region), but in other situations a bar chart with error bars might be used to display the mean response levels of a continuous variable (e.g. weight, concentration) and its standard error among several categories as an alternative to a box plot. E.g. a bar chart can be used to plot the frequencies of each adverse event in the AE data set (Figure 12), or a bar chart could be used to plot the mean BMI levels for male and female patients in the AFP data set (Figure 13).

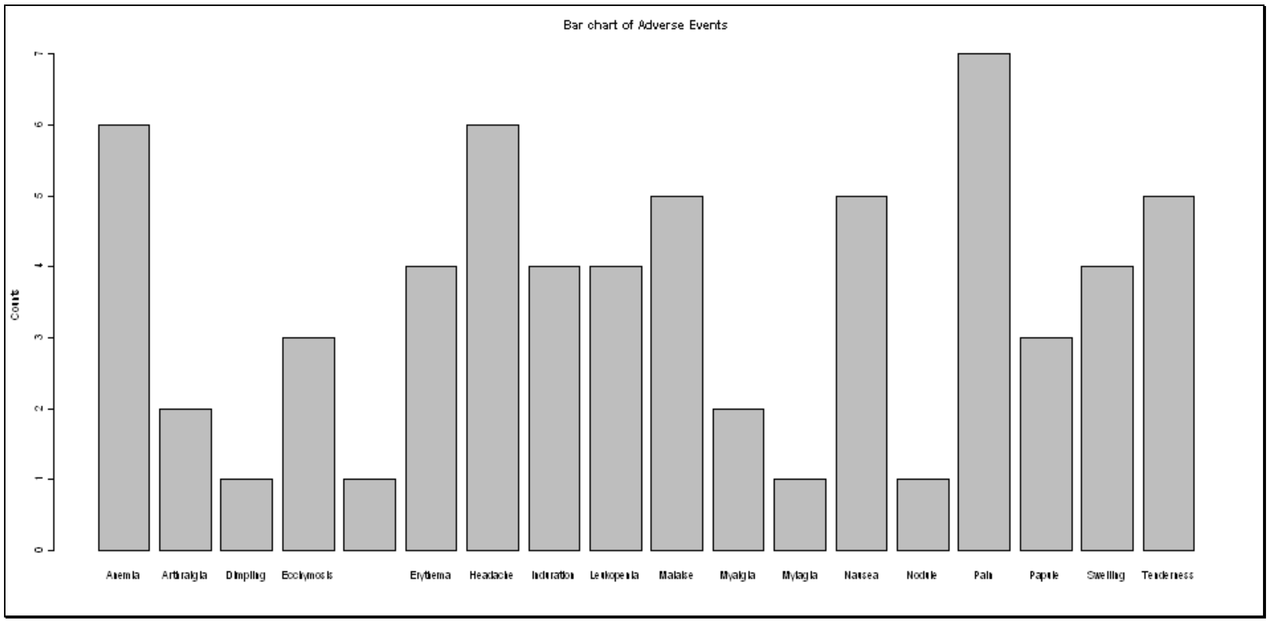


Figure 12. A bar chart of adverse events from the AE data set

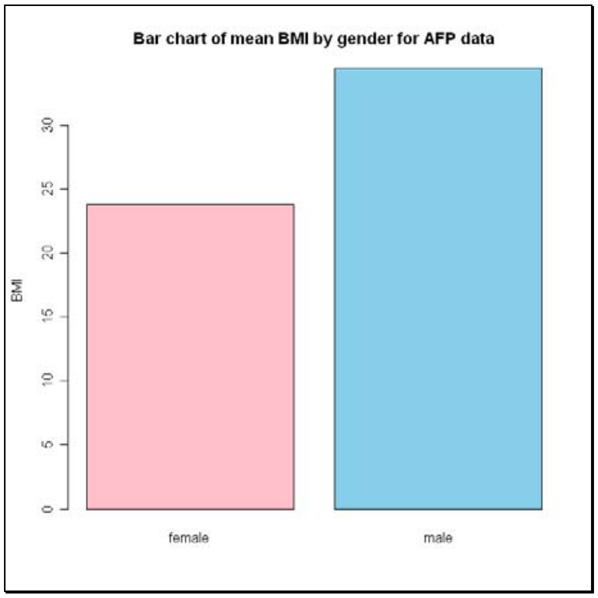


Figure 13. A bar chart of female and male BMI from the AFP data set.

```
> # Create a vector of counts for the AE bar chart
> counts <- summary(AE$Adverse.Event)
> # Define a main title for the bar chart
> main <- "Bar chart of Adverse Events"
> # Call the barplot() command for a bar chart of adverse events
> barplot(height=counts,main=main,ylab=”Counts”)
> #
> # compute mean BMI for male and female patients from AFP data
> BMI.females <-mean(AFP[AFP$gender=="female",5])
> BMI.males <-mean(AFP[AFP$gender=="male",5])
> mean.BMI <- c(BMI.females,BMI.males)
> # define labels for female and male bars
> names(mean.BMI) <- levels(as.factor(AFP$gender))
> # Define colors for female and male bars
> colors <- c("pink","sky blue")
> # Specify a main title for the bar chart graph
> main   <- "Bar chart of mean BMI by gender for AFP data"
> # Call the barplot() command for a bar chart of BMI responses
> barplot(height = mean.BMI,ylab="BMI", main=main,col=colors)
```

Before creating a bar chart of mean BMI levels for male and female patients from the AFP data, we first need to compute the individual BMI means for male and female patients using subscripting. Names were assigned to the vector of BMI means using the `names()` function, so the appropriate gender labels will show up below the male and female BMI data. Colors and a main title were defined for the chart, and the `barplot()` function was called with the appropriate options.

Note that more advanced bar charts of several categorical variables can be easily created with the `barchart()` command from the `lattice` package library. Multiple categorical variables can be summarized with the `table()` command, then the table of categorical variables is entered into the `barchart()` command for easy clustered, stacked or paneled bar charts. Notice that numeric variables will not work appropriately in the `barchart()` command. However, quick and easy bar charts with error bars can be created with the `bargraph.CI()` command from the `sciplot` package library. Other helpful packages may exist to create more variations on these types of bar charts.

2.1.5 Simple scatter plots and line plots

Scatter plots are used to display the relationship between two continuous variables that might be analyzed using linear regression or nonlinear regression models. E.g. an XY scatter plot might be used to examine the relationship between % body fat and weight (lbs) from the AE dataset. Line plots are often used to plot survival curves, probability density functions (PDFs), cumulative distribution functions (CDFs) and other continuous functions of interest. E.g. you might need a plot of the standard normal curve for a class lecture or a statistics textbook.

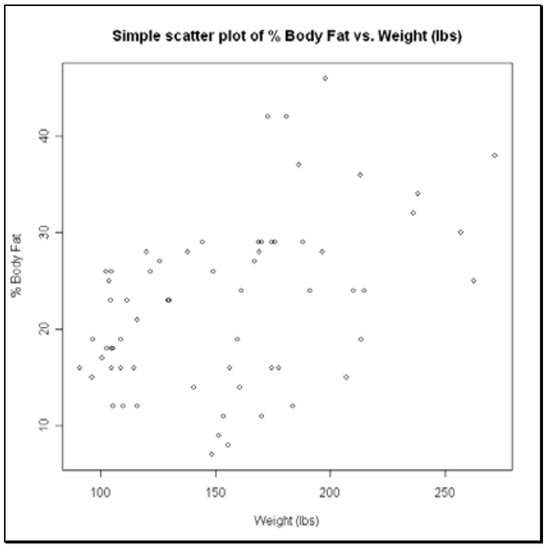


Figure 14. A scatter plot of % Body Fat vs Weight (lbs) from the AE data

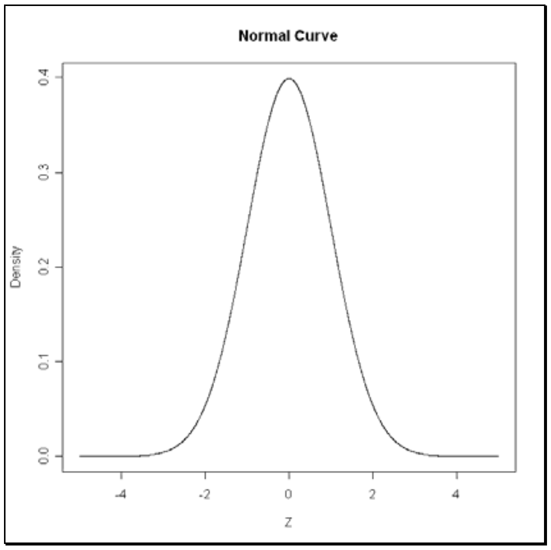


Figure 15. A plot of the (Gaussian) normal density function (mean = 0, sd = 1).

```
> # Define a main title for a scatter plot
> main <- "Simple scatter plot of % Body Fat vs. Weight (lbs)"
> # Simple scatter plot of % Body Fat vs. Weight
> plot(AE$Weight,AE$Percent.Body.Fat,xlab="Weight (lbs)",ylab="%
      Body Fat",main=main)
> #
> # Define a continuous sequence Z ranging from -5 to +5
> Z <- seq(from=-5,to=5,length=8000)
> # Define a sequence representing the density of a normal curve
> fZ <- dnorm(Z)
> # Plot a normal curve
> plot(Z,fZ,type="l",ylab="Density",main="Normal Curve")
```

A main title was defined for the XY scatter plot of % body fat vs. weight (lbs) from the AE data, before calling the `plot()` command with its `xlab` and `ylab` options to define the X- and Y-axis labels, respectively. Since the probability density of a standard normal distribution is really a function *f(x)*, two new variables `Z` and `fZ` were defined to create a line plot of the standard normal density. First, the variable `Z` was defined as a sequence of 8000 evenly spaced rational numbers from -5 to +5 using the `sequence()` command. Second, the variable `fZ` was defined as a sequence of 8000 numbers resulting from the function *f(x)* using the `dnorm()` command in R. Finally, a line plot was created from the `plot()` function by using the parameter `type =”l”`.

2.2 Custom Titles, Subtitles and Axes Labels

Most graphics procedures (e.g. `pie()`, `hist()`, `plot()`, ...) have some common parameters that allow users to add specific text for the main titles, subtitles and axes labels. There are additional commands that allow you to customize the look and feel of these labels for a more professional look. The following sections reveal some helpful tips about customizing the labels on a graph.

2.2.1 Adding and removing groups from a factor variable

Take a close look at the pie chart (Figure 7) and bar chart (Figure 12) created from the adverse events of the AE data. You may have noticed a possible typo in the data set, because the data contains two very similar groups “myalgia” and “mylagia”. The “mylagia” group is a typo, but can it be removed from the plot?

```
> # Examine the 18 levels of the Adverse.Event variable
> AE$Adverse.Event
 [1] Tenderness    Arthralgia    Mylagia       Erythema      Erythema      Anemia
Anemia
 ...
[57] Nausea        Headache      Nodule        Anemia        Swelling
Leukopenia    Elavated CH50
[64] Headache
18 Levels: Anemia Arthralgia Dimpling Ecchymosis Elavated CH50 Erythema Headache
Induration ... Tenderness
> # Store the list of variable names as a new variable
> new.labels <- levels(AE$Adverse.Event)
> # Verify the list still has 18 levels
> length(new.labels)
[1] 18
> # Use indexing to replace the “Mylagia” label with “Myalgia”
> new.labels[12] <- "Myalgia"
> # Assign these new labels to the levels of Adverse.Event
> levels(AE$Adverse.Event) <- new.labels
> # Verify Adverse.Event now has only 17 levels
> AE$Adverse.Event
```

```
 [1] Tenderness    Arthralgia    Myalgia       Erythema      Erythema      Anemia
Anemia
 ...
[57] Nausea        Headache      Nodule        Anemia        Swelling
Leukopenia    Elavated CH50
[64] Headache
17 Levels: Anemia Arthralgia Dimpling Ecchymosis Elavated CH50 Erythema Headache
Induration ... Tenderness
> # Redefine the counts to create a pie chart
> counts <- summary(AE$Adverse.Event)
> # Generate a new pie chart of Adverse Events
> pie(x=counts,main="Adverse Events",sub="Factor level typo 'Mylagia' has been
      corrected")
```

The first step of the process is to display the factor variable `AE$Adverse.Event` to view the number of factor levels and their names. There are 18 levels, including both the levels "Myalgia" and "Mylagia". Next, the `levels()` command is used on the right-hand side of the assignment arrow to define a new variable `new.labels`, which is a list of all 18 factor levels. Indexing is used to replace the 12th element of the `new.labels` variable, which contains the incorrect label "Mylagia". After the incorrect label has been replaced, the `levels()` command can be used on the left-hand side of the assignment arrow to re-define the factor level definitions of the `AE$Adverse.Event` variable. After re-defining the factor levels, the `AE$Adverse.Event` variable is displayed to reveal there are now only 17 factor levels. Finally, the `summary()` and `pie()` commands are used to generate a corrected pie chart of the data (Figure 16).

2.2.2 Changing fonts, colors and label sizes

Later in this manual, the `par()` command will be used to create some multi-panel figures and make other changes to multiple graphs. For now, you should read some of the help documentation for the `par()` command, because many graphing parameters in `par()` can be called within other graphing procedures like `pie()`, `hist()`, `plot()`, etc. Among other things, these graphing parameters can be used to change the fonts, colors and label sizes of the main title, subtitle, X- and Y-axis labels in pie charts (Figure 16), histograms (Figure 17) and other graphics (Figure 18).

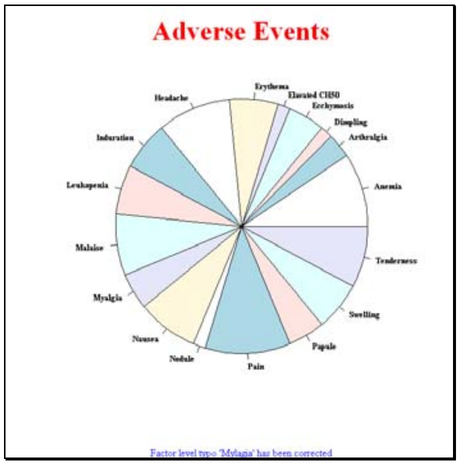


Figure 16. A pie chart with custom fonts and colors

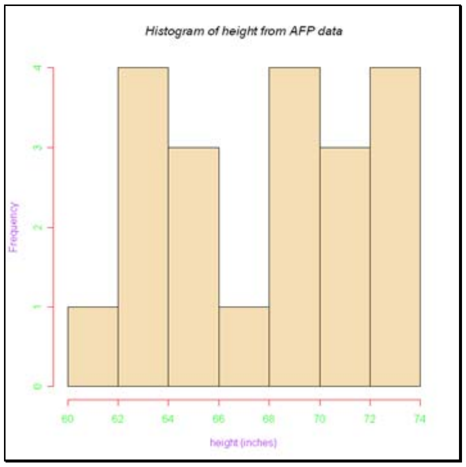


Figure 17. Histogram with custom fonts and colors

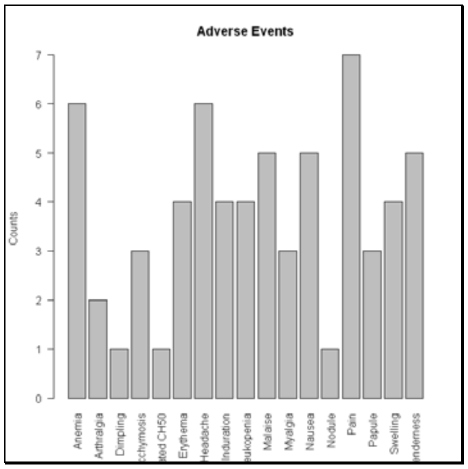


Figure 18. Bar chart with custom fonts and colors

```
> # Generate a pie chart from AE data with customized labels
> pie(x=counts, main=main, sub=subtitle, font=2, family="serif", cex=0.8,
      cex.main=2.6, col.main="red", col.sub="blue")
> # Generate a histogram from AFP data with customized labels
> hist(x=height, xlab=xlab, main=main, col="wheat", col.axis="green",
      col.lab="purple", fg="red",font.main=3)
> # Generate a barplot from AE data with customized labels
> barplot(counts,main=”Adverse Events”,ylab=”Counts”,las=2)
```

The `pie()`, `hist()` and `barplot()` commands were used to generate pie charts, histograms and bar charts in previous sections, but additional graphing parameters from the `par()` procedure are shown here to customize the graph labels. The `font` parameter specifies normal text (`font = 1`), boldface text (`font = 2`), italicized text (`font = 3`), bold and italicized text (`font = 4`) or symbol text (`font = 5`). The related parameters `font.axis`, `font.lab`, `font.main` and `font.sub` can assign text styles to the axis units, the axis labels, the main title or the sub title, respectively. The `family` parameter will change the font style for the entire figure with options for monospace font (`“mono”`), serif fonts (`“serif”`), sans-serif fonts (`“sans”`) and symbol fonts (`“symbol”`). Additional font families are available using Hershey vector fonts (e.g. `family = “HersheyScript”`). The `cex` parameter is used to shrink or enlarge the text by a percentage (e.g. `cex = 0.8` yields text at 80% its original size). Related parameters `cex.axis`, `cex.lab`, `cex.main` and `cex.sub` affect axis ticks, axis labels, main titles and sub titles, respectively. The `col` parameter typically affects objects in a graph (e.g. pie chart slices or histogram bars), but the `col.axis`, `col.lab`, `col.main` and `col.sub` parameters control font colors in the axis ticks, axis labels, main titles and sub titles. The `fg` parameter adjusts the color of the foreground, which typically includes the graph axes. Finally, the `las` parameter is used to adjust the orientation of axis tick labels (e.g. `las = 2` creates labels perpendicular to the x-axis).

2.2.3 Subscripts, superscripts and custom symbols

Sometimes, it will be necessary to use subscript or superscript text in the text of a graph or figure. For example, you may want to plot the residuals from a linear regression against $\log_{10}$-transformed predictor variable (Figure 19) or against the quadratic term from a polynomial regression (Figure 20). You may also need to use special characters, like Greek symbols or math symbols, mixed with regular text (Figure 21). These custom labels can be generated using `expression()` and `paste()` commands. In practice, the `expression()` command is used to generate mathematical equations for use as text labels in graphs, figures and reports. The `paste()` command is used to concatenate, or join, two or more strings of characters together to form a single character string. Together, these functions can be used to combine regular text with mathematical display styles, like superscript or subscript text, true fractions, Greek symbols, etc.

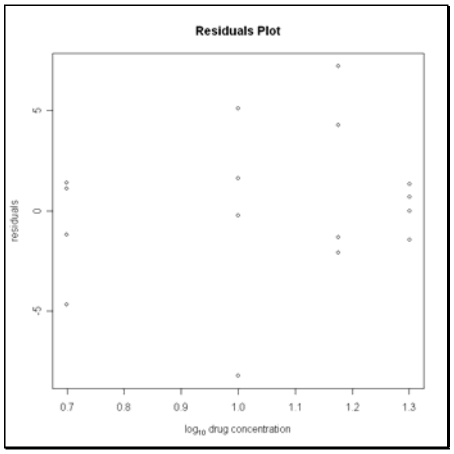


Figure 19. Scatter plot figure with subscript in X-axis label

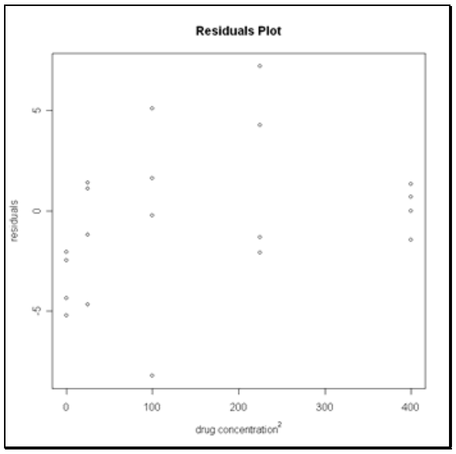


Figure 20. Scatter plot figure with superscript in X-axis label

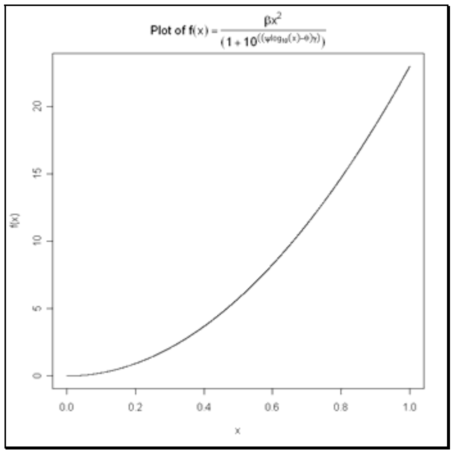


Figure 21. Line graph with Greek symbols and formulas in title.

```
> # Generate a vector of simulated "residuals"
> residuals <- rnorm(20,0,3.5)
> # Generate log10 transform of drug variable
> log10drug <- log(AFP$drug,10)
> # Generate drug-squared quadratic term
> drugSq    <- AFP$drug*AFP$drug
> # Residuals plot with subscript labels
> plot(log10drug, residuals, main="Residuals Plot",
      xlab=expression(paste(log[10]," drug concentration")))
> # Residuals plot with superscript labels
> plot(drugSq, residuals, main="Residuals Plot",
      xlab=expression(paste("drug concentration"^2)))
> # Generate sequence from 0 to 1
> x <- seq(from=0,to=1,length=8000)
> # Define a function f(x)
> f <- function(x) 23*x^2 / (1 + 10^((6*log10(x) - 15)*0.7))
> # Plot function with math formula in title
> plot(x,f(x),type="l",main=expression(paste("Plot of ",f(x) == frac(beta*x^2 ,
      (1 + 10^((psi*log[10](x) - theta)*gamma))))))
```

In the first graph (Figure 19), square brackets are used to specify the subscripted text within the `expression()` command and the `paste()` command is used inside the expression command to join the mathematical expression “$\log_{10}$” with the rest of the text string. In the second graph (Figure 20), the carrot symbol “^” identifies the superscripted text. The third graph (Figure 21) features a much more complicated mathematical formula in the `expression()` command. Note how the double equal sign “==” is used to create a regular equal sign within the mathematical expression and the function “frac(a,b)” is used to generate a large fraction graphic in the main title. Greek symbols are generated within the `expression()` command using the usual English phrases (e.g. θ = “theta”, π = “pi”).

2.3 Custom Color and Layout Options

The previous sections of this chapter described how to produce basic graphs and customize their labels, but often the graphic itself needs to be customized with new colors or a more informative layout. For example, you have already seen that histogram graphs can provide very different interpretations of the same sample of data just by changing the number of break points, or bins, in the histogram. A histogram with too few bins or too many bins may not be very informative. Similarly, the layout of the X- and Y-axes of a boxplot, bar chart, scatter plot or line plot figure can impact the effectiveness of a graph. For example, the Y-axis in the bar chart of mean BMI from the AFP data (Figure 13) is too short, so it is difficult to estimate the mean BMI for the male patients. You may have also noticed that the default layout can sometimes produce missing or incomplete axis ticks and labels in figures. For example, the bar charts of adverse events from the AE data (Figure 12 and Figure 18) either have missing horizontal labels or vertical labels that run off the page. Finally, the default coloring options for the pie chart of adverse events (Figure 7 and Figure 16) are not particularly eye-catching, but it seems difficult to specify 17 different colors by name for an effective layout. This section will describe options to change the layout and coloring of these figures.

2.3.1 Custom X- and Y-axes

There are several features to customize in the X- or Y-variable of a graph, including the minimum and maximum values of the axis and the scale of the numbered “major” ticks of an axis. Most of these features can be directly modified using the parameters from the `par()` command. For example, it would be easier to interpret the boxplot of mean BMI for male and female patients in the AFP data set if the Y-axis extended past 30 (Figure 13). It may be easier to interpret the XY scatter plot of weight and % body fat from the AE data if the X- and Y-axes had more tick marks to provide finer resolution of the individual weight and % body fat values. More complicated changes to the X- or Y-axes, like axis breaks or separate left and right Y-axes, may require new R package libraries.

```
> # Call the barplot() command for a bar chart of BMI responses
> barplot(height = mean.BMI,ylab="BMI", main=main,col=colors, ylim = c(0,50),
      yaxp=c(0,50,10))
> # Simple scatter plot of % Body Fat vs. Weight
> plot(AE$Weight,AE$Percent.Body.Fat,xlab="Weight (lbs)",ylab="% Body Fat",
      main=main,xlim=c(85,290),xaxp=c(100,275,7),ylim=c(5,50),yaxp=c(5,50,9))
```

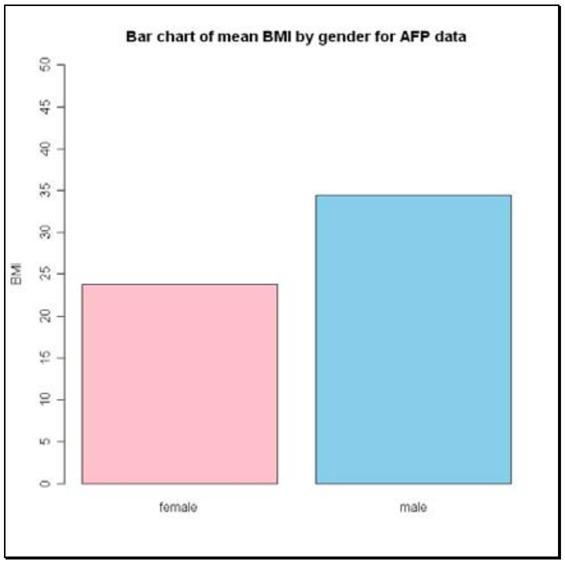


Figure 22. Bar chart with customized Y-axis limits and Y-axis tick intervals

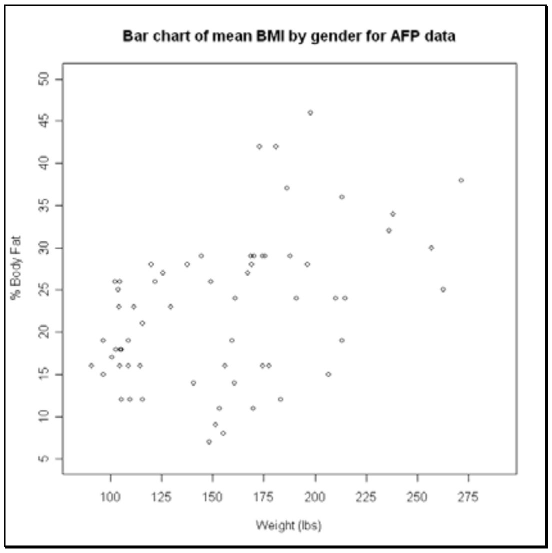


Figure 23. Scatter plot with customized X- and Y-axis limits and tick intervals

The `ylim` parameter was used to extend the Y-axis of the barplot (Figure 22) from BMI = 30 to BMI = 50. Notice both `ylim` and analogous `xlim` parameter require users to define the lower and upper limits of each axis with a vector input (e.g. `ylim = c(0,50)` is used to specify a lower limit of 0 and an upper limit of 50). Both the `xaxp` and `yaxp` parameters are used to add finer resolution tick marks to the XY scatter plot (Figure 23). The `xaxp` and `yaxp` parameters require users to specify the lower and upper limits of the tick marks and the number of ticks in between (e.g. `xaxp = c(100,275,7)` indicates that the ticks should start at 100 and end at 275 with 7 ticks in between). Notice the lower and upper limits of `xlim` and `xaxp` or `ylim` and `yaxp` can be different, if the tick marks should not start at the very beginning of an axis.

Some other features related to axis customization include the `par()` parameters `bg`, `bty`, `fg`, `tck`, `tcl`, `xlog` and `ylog`. The parameters `bg` and `fg` control the colors of the background and foreground of the graph, respectively. The `bty` parameter determines the style of the box drawn around the graph. The `tck` and `tcl` parameters adjust the length of the axis ticks relative to the size of the plotting region and the size of a single line of text, respectively. The `xlog` and `ylog` parameters are used to plot continuous variables on a $\log_{10}$-transformed axis.

2.3.2 Custom graph sizes and positions

Controlling the size of an image can be critical, especially if the image is going to be printed or exported to another software program for further editing. Explicit control over the size of your graph, the graph margins and other features will ensure your figures look professional and can be easily understood. For example, you may have noticed that the group labels of the adverse events variable of the AE data set did not have enough space in the previous two bar charts (Figure 12 and Figure 19). Alternatively, there may be too much white space in the pie chart figures (e.g. Figure 17). These problems can be fixed by changing the size of the figure and its figure margins.

The `par()` parameters controlling the size of the plotting region, the size of the graph and the size of the margins typically need to be specified with a `par()` command or `plot.new()` command before a specific graphing procedure is called with a command like `pie()`, `hist()` or `barplot()`. It may help to close all open graphs manually or with the `dev.off()` command, before you begin to resize a figure. Once all graphs are closed, you can adjust the size parameters with a `par()` command, then call the usual graphing commands to generate the figure in the customized space.

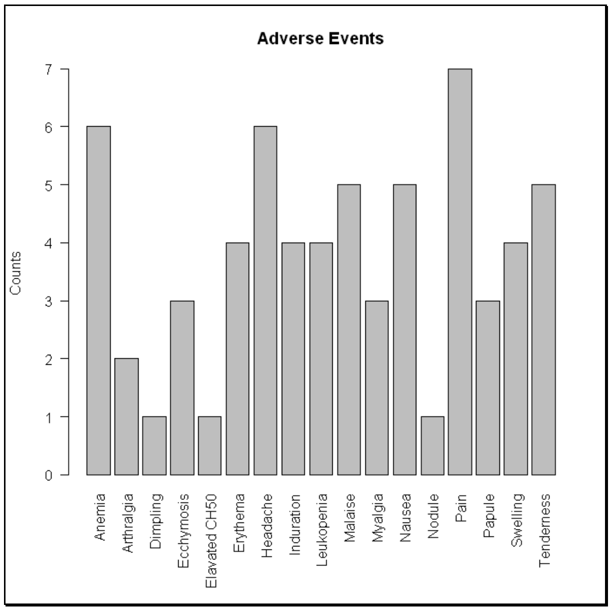


Figure 24. Bar chart with custom margin sizes

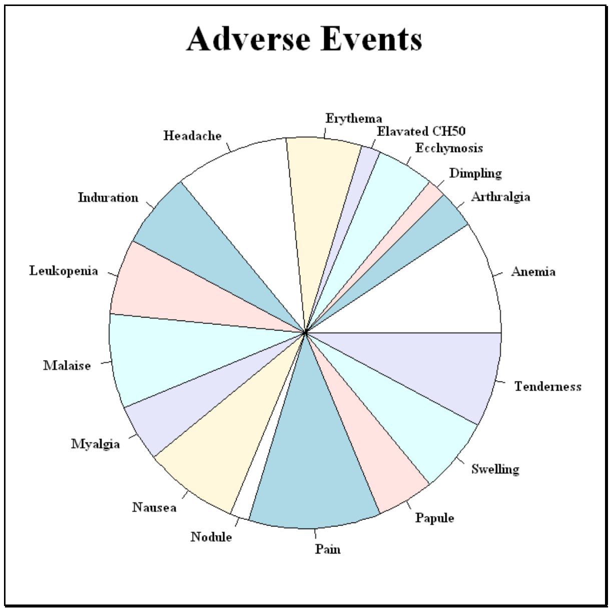


Figure 25. Pie chart with custom margin sizes

```
> # Adjust margins from mar = c(5,4,4,2) to mar = c(8,4,4,2)
> # where mar = c(bottom, left, top, right)
> par(mar=c(8,4,4,2))
> # plot bar chart as usual
> barplot(counts,main="Adverse Events",ylab="Counts",las=2)
> # Close the current graphic device window
> dev.off()
null device
          1
> # Adjust the margins for a pie chart
> par(mar=c(0.5,1,4,1))
> # Plot pie chart as usual
> pie(counts, main = main,cex.main=2.6,font=2,family="serif")
```

In this examples the inadequate x-axis margin of the bar chart (Figure 24) and the excessive white space of the pie chart (Figure 25) were both corrected using the `mar` parameter, which adjusts the size of the graph margins relative to the size of one line of text. The default setting is `mar = c(bottom = 5, left = 4, top = 4, right = 2)`, so the bar chart was given extra space in the bottom margin to accommodate the lengthy x-axis labels and the pie chart was given smaller margins all the way around to eliminate unnecessary white space. Margin space can also be specified in inches with the `mai` parameter or by an expansion factor with the `mex` parameter. Another alternative would be to adjust the size of the graph and plotting region using the `fig` or `fin` parameters.

2.3.3 Color gradient functions and translucent colors

Previous examples have shown that `par()` and other R graphics procedures allow users to change the colors of almost every element in a graph. However, color choices must be specified at the command line, so it is important to know the names of all the possible color choices. If you enter the command `colors()` at the R command line, you will see a list of 657 available color choices. Colors range from simple primary colors (e.g. `"red"`, `"blue"`, `"yellow"`) to subtle color gradients (e.g. `"gray0"` to `"gray99"`) and sophisticated palette choices (e.g. `"azure"`, `"burlywood"`, `"chartreuse"`, etc.). Each of these color choices must be specified by name, but they should provide more than enough options for most users.

Some R procedures, like the `pie()` command, will apply default color choices to your graph if nothing is specified by the user. The `grDevices` package library includes a handful of functions that will allow you to generate a known gradient of colors for a given number of groups. For example, the `rainbow(n=10)` command will generate a list of 10 color choices from a gradient of all possible "rainbow" colors from red to orange to yellow to green to blue to purple and back to red again (Figure 26). The `heat.colors()` command will similarly generate a list of colors from yellow to orange to red for use in heatmap figures. These "high-level" color gradient functions use the more generic `rgb()`, `hsv()` or `hcl()` functions to generate colors using "red-green-blue", "hue-saturation-value" or "hue-chroma-luminance" methods. Advanced users may choose to call these `rgb()`, `hsv()` or `hcl()` colors directly, either by calling the functions themselves or by calling their output values (Figure 27 and Figure 28). Note the `rgb()`, `hsv()` or `hcl()` commands include an `alpha` parameter that is used to specify transparent colors.

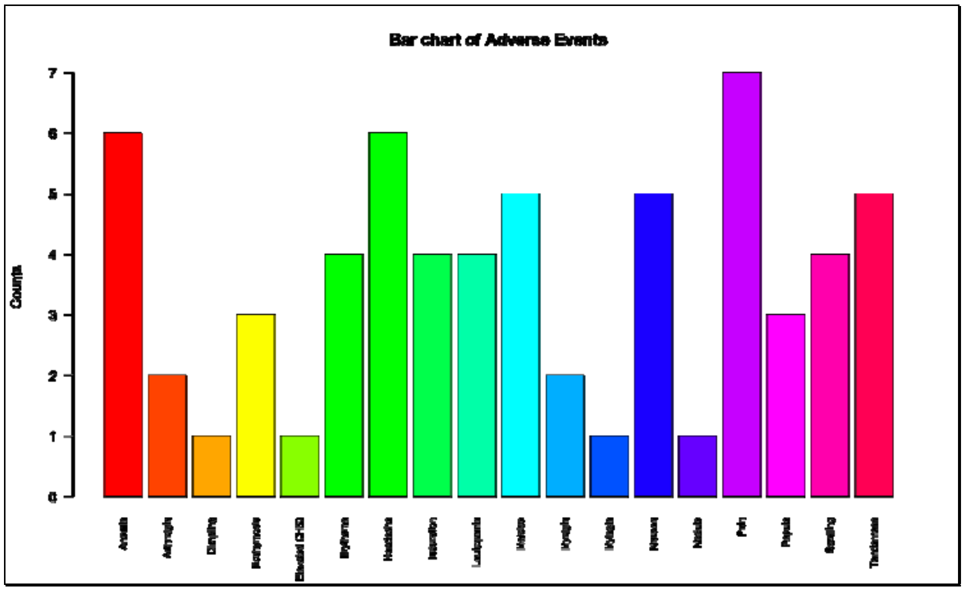


Figure 26. A bar chart of adverse events from the AE data set using `rainbow()` colors

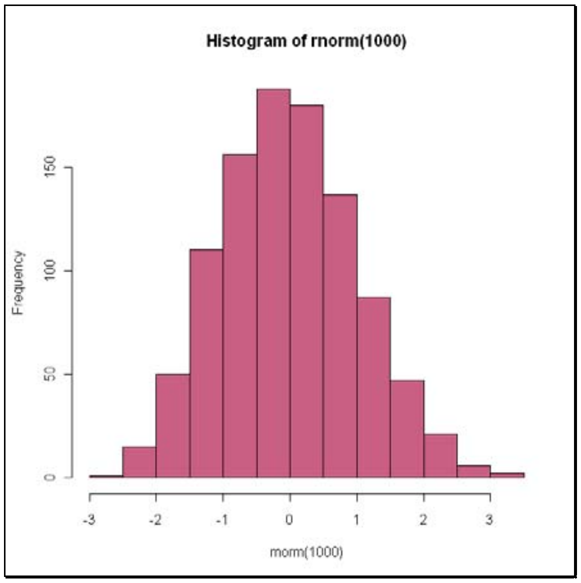


Figure 27. Histogram with `rgb()` color.

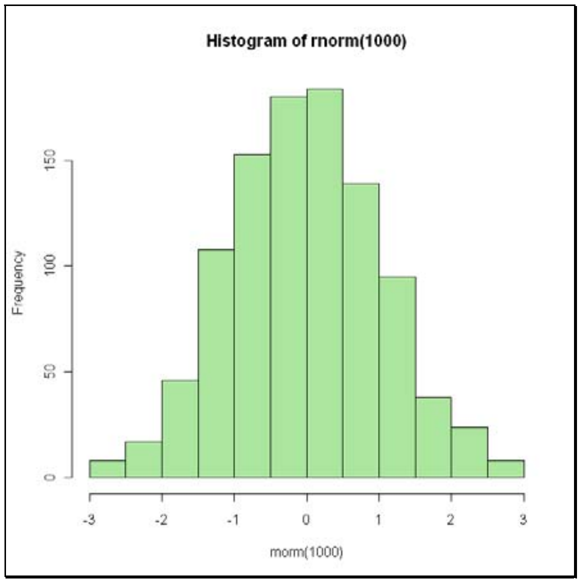


Figure 28. Histogram with `hsv()` color

```
> # Use the rainbow() command to generate rainbow colors
> barplot(height=counts,col=rainbow(18),cex.names=0.7,las=2,...)
> # Use rgb() to generate a custom color for a histogram
> hist(rnorm(1000),col=rgb(red=0.7,green=0.1,blue=0.3,alpha=0.7,
      maxColorValue=1))
> # View the value name of the custom rgb() color
> rgb(red=0.7,green=0.1,blue=0.3,alpha=0.7,maxColorValue=1)
[1] "#B21A4CB2"
> # Call the custom rgb() color by its value name
> hist(rnorm(1000),col="#B21A4CB2")
> # View the value name of a custom hsv() color
> hsv(h=0.3,s=0.7,v=0.8,gamma=1,alpha=0.5)
[1] "#5ACC3D80"
> # Call the custom hsv() color by its value name
> hist(rnorm(1000),col="#5ACC3D80")
```

Ch. 3. Multi-step Graphics with Figure Legends

Most graphs and figures can be produced with a single R command. However, some complicated graphs may require multiple statements to overlap new points or lines, to add figure legends or to add features like text labels. This section provides examples of complicated figures that require multiple commands.

3.1 Overlay New Content in the Same Graph

Often an R graphic will require the user to overlay different types of content on the same graph or figure. Often users will need to overlay very similar types of content onto the same graph. For example, data from several groups might be displayed on the same scatter plot figure using points with different colors or shapes. Other figures may require users to overlay related types of information. E.g. Error bars representing the standard error of the mean might be added to a bar chart displaying the mean BMI of male and female patients. Sometimes, users may want to overlay entirely different types of content. E.g. A user might want to display the fitted density of a standard normal curve over a histogram to show that the variable is normally distributed. The next few examples will demonstrate some general principles for creating overlay figures.

3.1.1 Overlay two different graphs using `par(new=TRUE)`

In previous sections, the `hist()` command was used to generate a histogram of a single sample and the `plot()` command was used to generate a plot of the density from a normal distribution. The `par(new=TRUE)` command is used to overlay one figure on top of another, which can be used to add a fitted normal distribution over a histogram (Figure 29) or to plot two histograms side-by-side using translucent colors (Figure 30).

```
> # Generate two number sequences from -4 to +4
> aa <- seq(from=-4,to=4,length=8000)
> bb <- seq(from=-4,to=4,length=40)
> # Create a histogram from random normal data
> hist(rnorm(1000),col="#B21A4CB2",breaks=bb,freq=FALSE,xlab="X",ylim=c(0,0.5))
> # Call par(new=TRUE) to overlay a new figure on the histogram
> par(new=TRUE)
> # Overlay the density of a standard normal distribution
> plot(aa,dnorm(aa),xlim=c(-4,4),ylim=c(0,0.5),type="l", xlab="",ylab="")
> # Close the current graphics window
> dev.off()
```

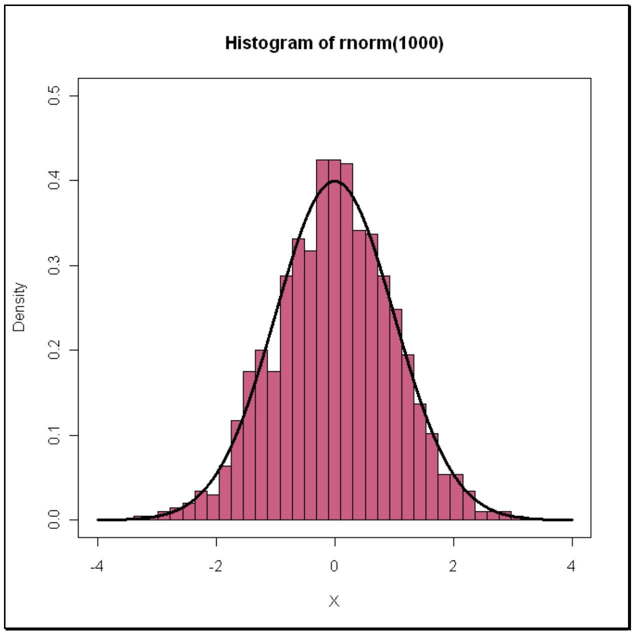


Figure 29. Histogram with fitted normal density

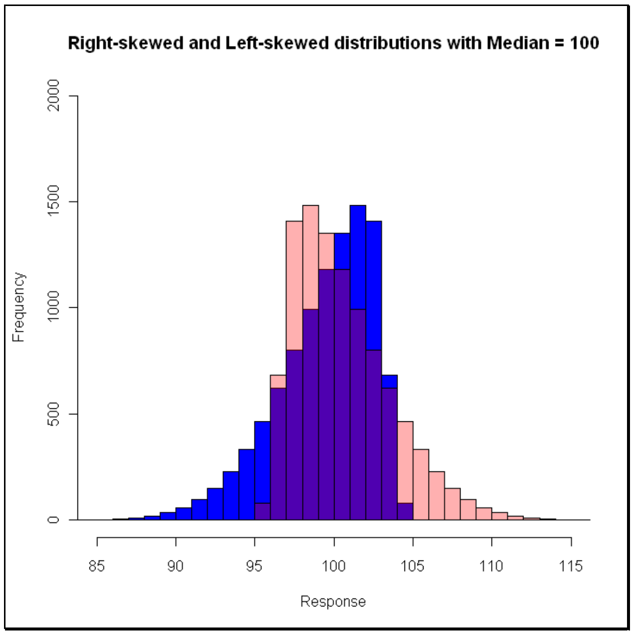


Figure 30. Two histograms overlay figure.

```
> # Load the mnormt and sn package libraries to generate
> # skewed normal data for two new histograms
> library(mnormt)
> library(sn)
> # Generate a sequence from 0 to 1
> uu <- seq(from=0,to=1,length=10000)
> # Create random samples from right- and left-skewed normal data
> rr <- qsn(uu,96.6276,5,8)
> ll <- qsn(uu,103.37245,5,-8)
> # Generate a histogram from the left-skewed data
> hist(ll,breaks=30,xlim=c(85,115),ylim=c(0,2000),col="blue", main="Right-skewed
    and Left-skewed distributions with Median = 100",xlab="Response")
> # Call par(new=TRUE) to overlay a new figure on the histogram
> par(new=T)
> # Generate a histogram from the right-skewed data
> hist(rr,breaks=30,xlim=c(85,115),ylim=c(0,2000),col="#FF00004F",main="",xlab="",
      ylab="")
```

There are several important considerations to make when one graph is overlaid on another. You must make sure that both graphs share the same X- and Y-axis ranges and tick intervals, so the two graphs match up appropriately. You also need to apply main titles, X- and Y-axis labels and subtitles to only one of the two graphs. With this in mind, two sequences from -4 to +4 were created to define the domain of the normal density (`aa`) function and the break points of the histogram (`bb`). The break points define the range of the X-axis for the histogram and matching `ylim` parameters were used to ensure both the graphs would match properly. The `par(new = TRUE)` command was called after the `hist()` command, to indicate that the subsequent `plot()` command was displayed on top of the existing histogram graphic. The result is a histogram with a fitted normal density (Figure 29). The `dev.off()` command was entered to close the current graphic window, so the next graph can be created.

The `mnormt` and `sn` R package libraries were installed and and loaded with the library command to generate the figure with overlapping histograms (Figure 30). The `qsn()` command from the `sn` package was used to generate histograms representing both left- and right-skewed normal distributions. Notice `ll` and `rr` do not represent random samples from these distributions, but instead they represent histograms of the complete density functions generated using a uniform sequence of 10,000 numbers from 0 to 1 (`uu`) and the quantile function of the skewed normal distribution, `qsn()`. The histogram of `ll` was created first, followed by a `par(new=TRUE)` statement and the histogram of `rr` with a transparent color from the `rgb()` command. This figure shows left- and right-skewed normal distributions that both have median = 100, which can be used as counter example to generate a significant Wilcoxon test statistic even though both samples have the same median.

3.1.2 Add error bars to a bar chart using `segments()` or `arrows()`

In section 2.1.4, bar charts were used to show the frequency of outcomes for categorical variables, like the adverse events variable in the AE data set, or the mean response for continuous variables, like the mean BMI levels for male and female patients in the AFP data set. A bar chart of mean or median response levels is typically used to accompany a student's t-test, Wilcoxon test or analysis of variance (ANOVA), but these figures are much more informative if they include error bars of some kind. The base packages in R do not compute error bars automatically, so users will need to compute their own error statistics and manually overlay the error bars on an existing `barplot()` graphic using the `segments()` or `arrows()` command (Figure 31).

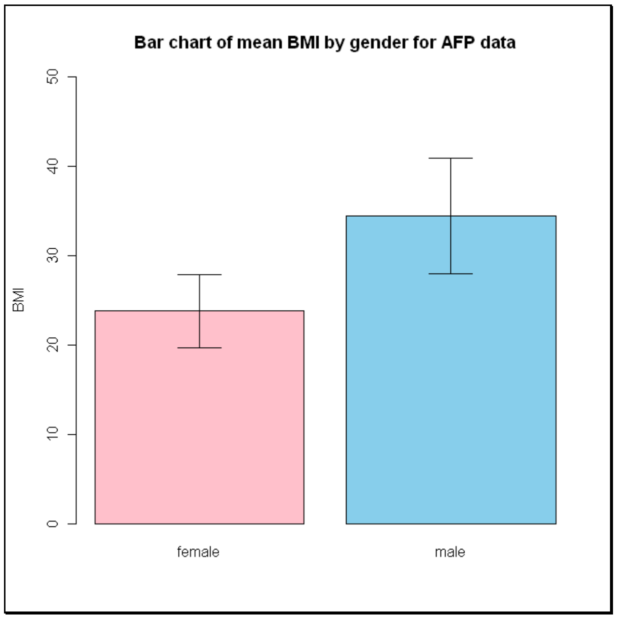


Figure 31. Bar chart with custom error bars

```
> # compute std dev of BMI for male and female patients
> sd.females <- sd(AFP[AFP$gender=="female",5])
> sd.males   <- sd(AFP[AFP$gender=="male",5])
> sd.BMI     <- c(sd.females,sd.males)
> # call the barplot() command for a bar chart of BMI
> mp <- barplot(height=mean.BMI,ylab="BMI",ylim=c(0,50),main=main,col=colors)
> # define the starting and ending points of each arrow
> X0 <- X1 <- mp
> Y0 <- mean.BMI - sd.BMI
> Y1 <- mean.BMI + sd.BMI
> # call the arrows command to overlay error bars over barplot
> arrows(X0,Y0,X1,Y1,code=3,angle=90)
```

Again, creating a overlay figure in R requires a little forethought. The variable `mean.BMI` was defined earlier in section 2.1.4 to compute the mean BMI response for both male and female patients. Here, similar methods are used to compute the standard deviations of BMI for both male and female patients and store them as `sd.BMI`. Next, the `barplot()` command is used to generate the bar chart figure, but this time the figure is also stored as the variable `mp` so the midpoints of each bar can be stored. Each error bar will need starting and stopping locations on both the X- and Y-axis. Here the starting location (`X0`) and stopping location (`X1`) for the X-axis are both defined as the midpoints (`mp`) from the barplot procedure, while the starting and stopping points on the Y-axis (`Y0` and `Y1`) are computed as one standard deviation above and one standard deviation below the mean BMI for male and female patients, respectively. Finally, error bars are plotted using the starting and stopping locations in the `arrows()` or `segments()` commands. The `arrows()` command is used to produce capped whiskers, with the `angle = 90` parameter set to produce square whisker caps. The default `angle` parameter will produce arrowhead caps. The `segments()` command produces whiskers with no end caps.

3.1.3 Add regression lines using `abline()` or `plot()`

In section 2.1.5, the `plot()` command was used to produce a simple scatter plot of % Body Fat vs. Weight from the AE data set. Researchers might want to analyze this kind of data using simple linear regression or nonlinear regression methods. Without exploring the statistical details of these models, the next examples will demonstrate how regression lines can be added to a scatter plot graphic.

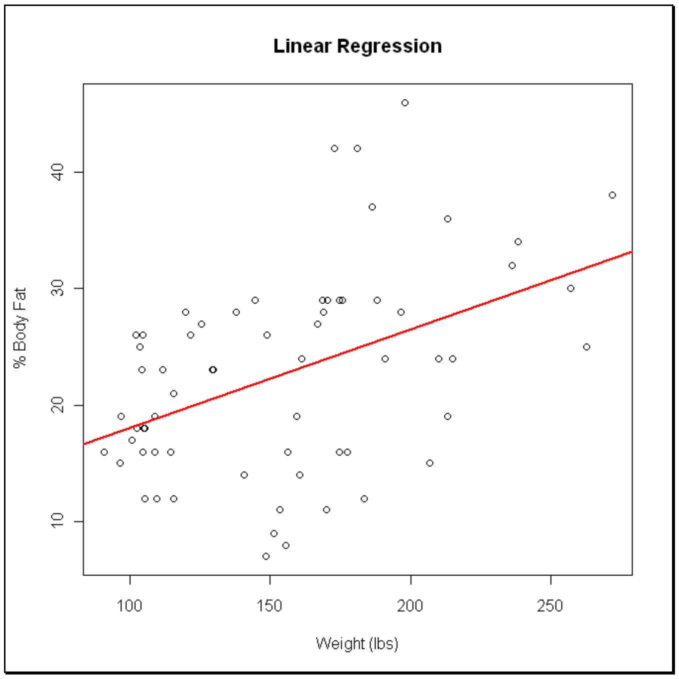


Figure 32. Scatter plot with linear regression line

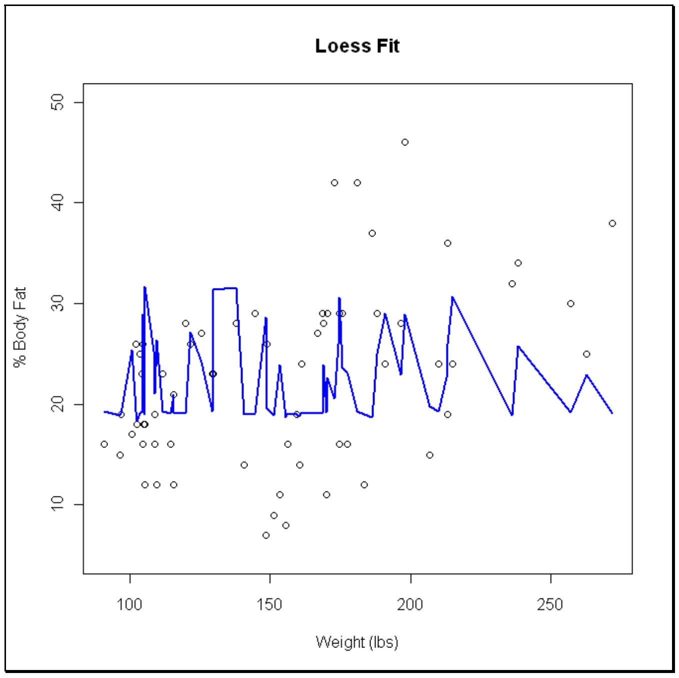


Figure 33. Scatter plot with nonlinear curve fit

```
> # Estimate a linear regression fit using lm()
> reg.line <- lm(Percent.Body.Fat ~ Weight,data=AE)
> # Create a simple scatter plot of % Body Fat vs Weight
> plot(AE$Weight,AE$Percent.Body.Fat,xlab="Weight (lbs)",ylab="% Body Fat",
      main="Linear Regression")
> # Add a linear regression line using abline()
> abline(reg.line,col="red",lwd=2)
> # Close the graphics device
> dev.off()
> # Estimate a smoothed loess fit
> loess.line <- loess(Percent.Body.Fat~Weight,data=AE,span=0.85)
> # Create a simple scatter plot of % Body Fat vs Weight
> plot(AE$Weight,AE$Percent.Body.Fat,xlab="Weight (lbs)",ylim=c(5,50),
      ylab="% Body Fat",main="Loess Fit")
> # Add the loess fit to the scatter plot
> par(new=TRUE)
> plot(sort(AE$Weight),predict(loess.line),ylim=c(5,50),type="l",lwd=2,col="blue",
      xlab="",ylab="")
```

The linear model procedure `lm()` is used to fit a simple linear regression to the continuous response variable % Body Fat and the continuous predictor variable Weight from the AE data set (Figure 32) and the results of the analysis are stored as `reg.line`. A scatter plot of % Body Fat vs. Weight is created using `plot()` and a regression line is added using the `abline()` command. There was no need to use a `par(new=TRUE)` command, since `abline()` was called immediately after the `plot()` command. The `abline()` function is specially designed to add straight lines to the most recent plot, using only the slope and Y-intercept of the line as its inputs. Using the `par(new=TRUE)` and `plot()` commands would have been much more difficult for this simple regression line.

Fitting a curved trend to the scatter plot is a little more difficult (Figure 33). The `loess()` command is used to fit a local polynomial regression to the data and the results are stored as `loess.line`. The statistical details of this procedure are not particularly important, except that it produces a curved fit to the data. The `plot()` procedure is used to produce the initial scatterplot, then `par(new=TRUE)` is called to add the regression line over the current graphic. Finally, a second `plot()` command is called to print the loess line, using the sorted Weight variable as the X-values for the curve and the predicted values from the stored loess.line variable as the Y-values for the curve.

3.1.4 Plot multiple groups using `points()` or `lines()`

Looking at the simple scatter plot of % Body Fat vs. Weight, it might be fair to ask whether additional variables like Gender or Region could affect the relationship between % Body Fat and Weight. It may be useful to separate the Gender or Region groups on the scatter plot to look for differences in the relationship between % Body Fat and Weight among these groups. Multiple groups can be added to a scatter plot graph one-at-a-time using the `points()` command. Groups can be separated by using multiple colors, multiple symbol types, multiple symbol sizes or all of the above.

```
> # Create main title and other features
> main.gender = "Percent Body Fat versus Weight by Gender"
> main.region = "Percent Body Fat versus Weight by Region"
> xlab = "Weight"
> ylab = "% Body Fat"
> # Create a scatter plot of the female data in pink
> plot(AE[AE$Gender=="Female",5],AE[AE$Gender=="Female",6],xlim=c(90,350),
      ylim=c(0,50),main=main.gender,xlab=xlab,ylab=ylab,col="pink")
> # Overlay the male data points in sky blue with points()
> points(AE[AE$Gender=="Male",5],AE[AE$Gender=="Male",6],col="sky blue")
```

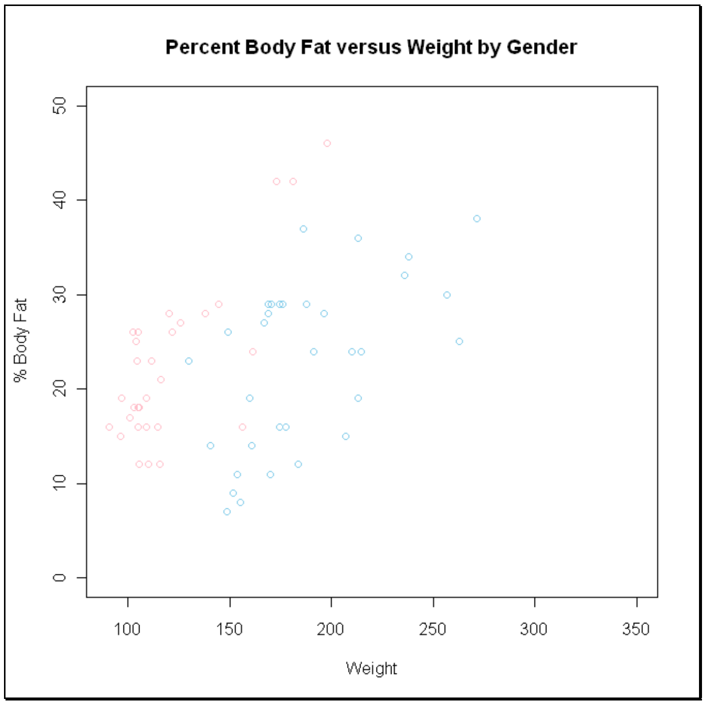


Figure 34. Gender groups distinguished by colors

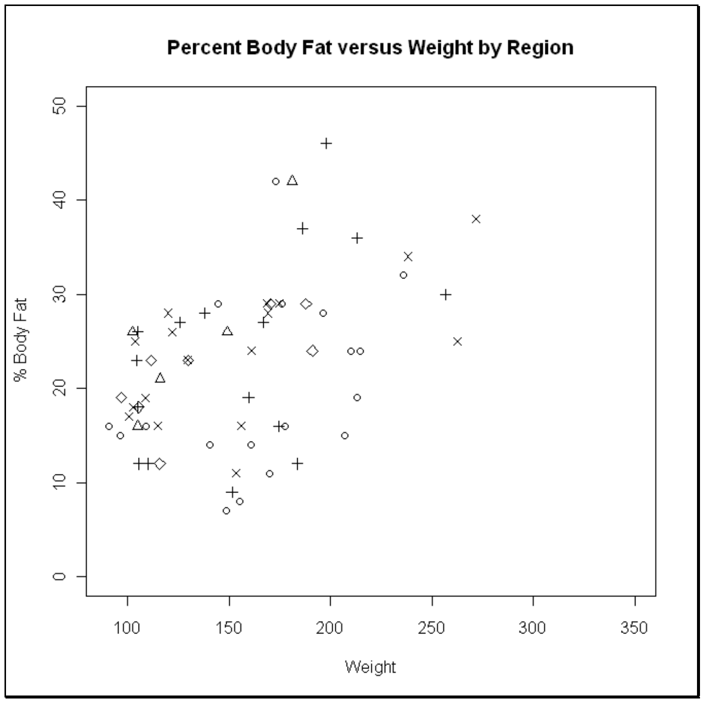


Figure 35. Region groups distinguished by symbols

```
> # Create a scatter plot of the Southwest region data
> plot(AE[AE$Region=="Southwest",5],AE[AE$Region=="Southwest",6],xlim=c(90,350),
      ylim=c(0,50),main=main.region,xlab=xlab,ylab=ylab)
> # Overlay data from other regions using new symbol types: pch
> points(AE[AE$Region=="Northwest",5],AE[AE$Region=="Northwest",6],pch=2)
> points(AE[AE$Region=="Midwest",5],AE[AE$Region=="Midwest",6],pch=3)
> points(AE[AE$Region=="Northeast",5],AE[AE$Region=="Northeast",6],pch=4)
> points(AE[AE$Region=="Southeast",5],AE[AE$Region=="Southeast",6],pch=5)
```

Just like the `abline()` command in section 3.1.3, the `points()` command can be used to add new data points to the existing graphic. To separate data by gender, plot the female data in pink using the `col` parameter in the `plot()` command and add the male data in sky blue using the `col` parameter in the `points()` command (Figure 34). To separate the data by region, first plot the data from the Southwest region using the default symbol and then plot the remaining regions with new symbol types using the `pch` parameter (Figure 35). The symbol size can also be adjusted with the `cex` parameter, but it should be used carefully because it also adjusts the size of all characters added to the plot.

Groups can also be separated on a line graph using multiple colors or multiple line types using the `lines()` or `ablines()` commands. Similar to the `points()` command, the `lines()` command can be used wherever `plot(type="l")` would be used. E.g. to plot probability density functions (Figure 36), nonlinear regressions, etc. Similarly, `abline()` can be used multiple times to plot several regression lines (Figure 37).

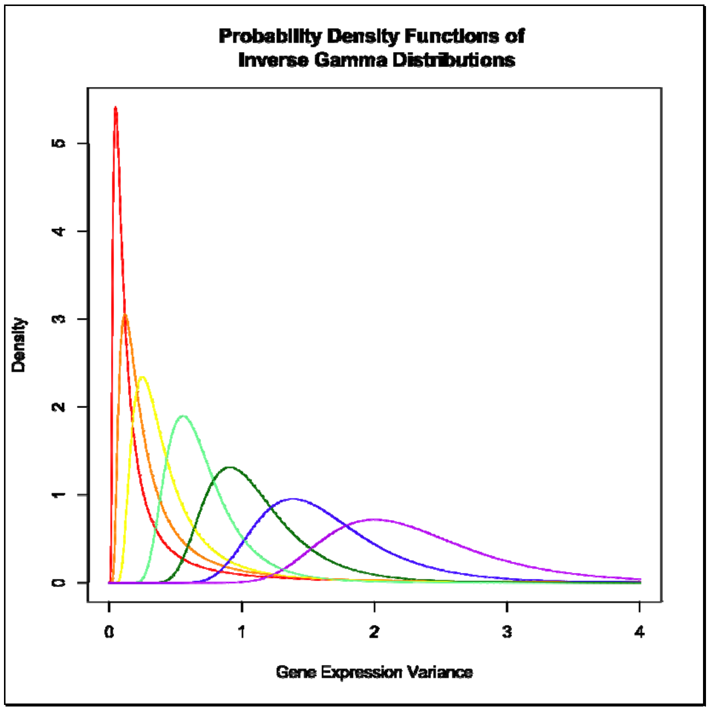


Figure 36. Seven inverse gamma density curves

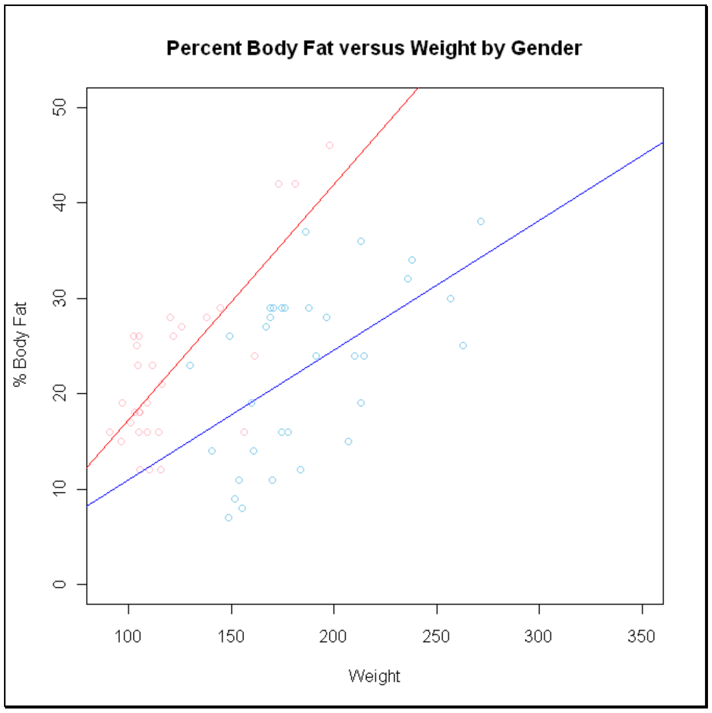


Figure 37. Two-color scatter plot with two line fits

```
> # Upload required libraries
> library(MCMCpack)
> # Create a sequence of numbers from 0 to 4
> x <- seq(from = 0, to = 4, length = 8000)
> # Plot the first probability density function
> plot(x,dinvgamma(x,1.0,0.1),type="l",col="red",xlab="Gene Expression Variance",
      ylab="Density",main="Probability Density Functions of \n Inverse Gamma
      Distributions")
> # Plot additional probability density functions
> lines(x,dinvgamma(x,1.5,0.3),col="dark orange")
> lines(x,dinvgamma(x,3.0,1.0),col="yellow")
> lines(x,dinvgamma(x,8.0,5.0),col="light green")
> lines(x,dinvgamma(x,10.0,10.0),col="dark green")
> lines(x,dinvgamma(x,12.0,18.0),col="blue")
> lines(x,dinvgamma(x,14.0,30.0),col="purple")
>
> # Fit two linear regression models
> AE.female  <- AE[AE$Gender == "Female",]
> AE.male    <- AE[AE$Gender == "Male",]
> reg.female <- lm(Percent.Body.Fat ~ Weight,data=AE.female)
> reg.male   <- lm(Percent.Body.Fat ~ Weight,data=AE.male)
> # Create a scatter plot of the female data in pink
> plot(AE[AE$Gender=="Female",5],AE[AE$Gender=="Female",6],xlim=c(90,350),
      ylim=c(0,50),main=main,xlab=xlab,ylab=ylab,col="pink")
> # Overlay the male data points in sky blue with points()
> points(AE[AE$Gender=="Male",5],AE[AE$Gender=="Male",6],col="sky blue")
> # Overlay two linear regression lines
> abline(reg.female,col="red")
> abline(reg.male,  col="blue")
```

3.2 Figure Legends and Overlaid Text

Many of the complicated graphics from subchapter 3.1 would require a figure legend to identify multiple groups on the same plot. Figure legends can be added to almost any figure and groups can be separated by color, line type and symbol type features from the graph. Additionally, text labels can be added to specific locations on any graph to label individual data points, such as outliers or specific patients.

3.2.1 Figure legends

In the previous sections, `barplot()`was used to create a bar chart of mean male and female BMI with standard deviation error whiskers. Both the `plot()` and `points()` commands were used to create scatter plot figures with separate colors and symbols for the gender and region variable, respectively. Figure legends could be used in both graphs to help readers distinguish between the groups on the plots.

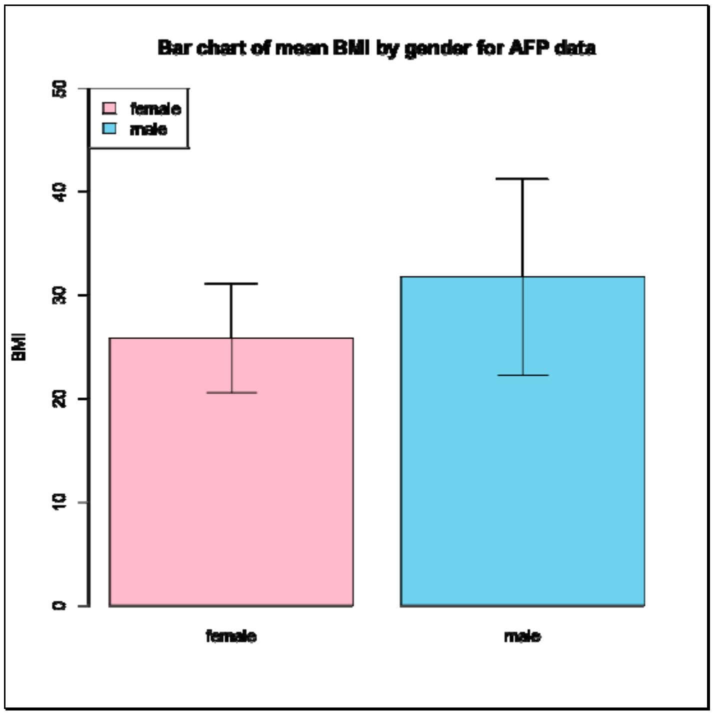


Figure 38. Bar chart with figure legend

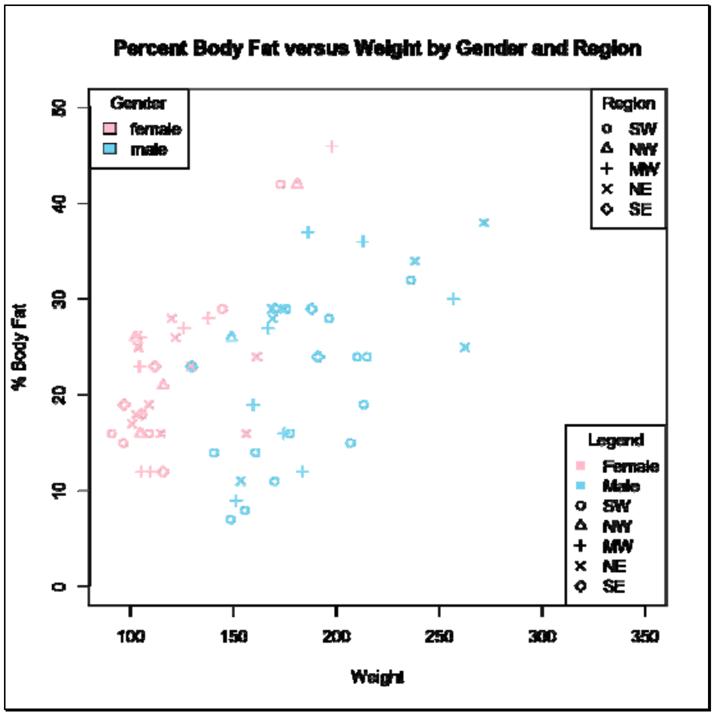


Figure 39. Scatter plot with three figure legends

```
> # Call the barplot() command for a bar chart of BMI
> mp <- barplot(height=mean.BMI,ylab="BMI",ylim=c(0,50),main=main,col=colors)
> # Define and plot the error whiskers
> arrows(X0,Y0,X1,Y1,code=3,angle=90)
> # Draw the figure legend
> legend(x="topleft",legend=c("female","male"),fill=c("pink","sky blue"))
> # Create indices for each plotted symbol
> indx.FSW <- AE$Gender == "Female" & AE$Region == "Southwest"
> indx.MSW <- AE$Gender == "Male"   & AE$Region == "Southwest"
> indx.FNW <- AE$Gender == "Female" & AE$Region == "Northwest"
> indx.MNW <- AE$Gender == "Male"   & AE$Region == "Northwest"
> indx.FMW <- AE$Gender == "Female" & AE$Region == "Midwest"
> indx.MMW <- AE$Gender == "Male"   & AE$Region == "Midwest"
> indx.FNE <- AE$Gender == "Female" & AE$Region == "Northeast"
> indx.MNE <- AE$Gender == "Male"   & AE$Region == "Northeast"
> indx.FSE <- AE$Gender == "Female" & AE$Region == "Southeast"
> indx.MSE <- AE$Gender == "Male"   & AE$Region == "Southeast"
> # Call the plot() and points() functions to generate figure
> plot(AE[indx.FSW,5],AE[indx.FSW,6],xlim=c(90,350), ylim=c(0,50),main=main,
      xlab=xlab,ylab=ylab,col="pink")
> points(AE[indx.MSW,5],AE[indx.MSW,6],col="sky blue")
> points(AE[indx.FNW,5],AE[indx.FNW,6],col="pink",pch=2)
> points(AE[indx.MNW,5],AE[indx.MNW,6],col="sky blue",pch=2)
> points(AE[indx.FMW,5],AE[indx.FMW,6],col="pink",pch=3)
> points(AE[indx.MMW,5],AE[indx.MMW,6],col="sky blue",pch=3)
> points(AE[indx.FNE,5],AE[indx.FNE,6],col="pink",pch=4)
> points(AE[indx.MNE,5],AE[indx.MNE,6],col="sky blue",pch=4)
> points(AE[indx.FSE,5],AE[indx.FSE,6],col="pink",pch=5)
> points(AE[indx.MSE,5],AE[indx.MSE,6],col="sky blue",pch=5)
> legend(x="topleft",legend=c("female","male"),fill=c("pink","sky blue"),
      title="Gender")
> legend(x="topright",legend=c("SW","NW","MW","NE","SE"),pch=1:5,title="Region")
> legend(x="bottomright",legend=c("Female","Male","SW","NW","MW","NE","SE"),
         col=c("pink","sky blue",rep("black",5)),pch=c(15,15,1:5),title="Legend")
```

The figure legend for the bar chart is easy to create using the `legend()` command (Figure 38). The `legend()` command has parameters `x` and `y` that are used to specify the exact location of the legend within the plotting region (e.g. `x` = 15 and `y` = 200 would place the figure legend at (x,y) = (15,200) in the graph, if it exists). While this precise level of control can be useful, many user will prefer to use shortcut locations like `x = “topleft”` or `x = “bottomright”` to place the legend in one of nine standard locations. See the Details section of `help(legend)` in R for more information. The `legend` parameter of the `legend()` command is used to specify the group labels in the legend. Note that you can choose to leave certain groups off the legend, or you can add notations for groups that are not present in the figure, if needed. Finally, the `fill` parameter is used to create filled boxes representing the colors of the male and female groups in this graph.

Note this new scatter plot graph is a little more complicated than the example in section 3.1.4 above (Figure 39). The first step was to create a series of ten index variables to identify individuals from each region and each gender, so these groups can be separated by symbol types and colors on the final plot. Each index was created using two inequalities evaluated simultaneously using the ampersand symbol `&`. The initial `plot()` command was only called for the female patients from the Southwest region, plotting with the default symbol and pink coloring. Remember, it is important to set reasonable x-axis and y-axis limits for this first plot with the `xlim` and `ylim` parameters, because all of the remaining groups will be plotted over this first figure. If `xlim` and `ylim` are chosen poorly, the remaining groups might not be visible. The next nine `points()` commands are used to add the remaining data to the plot, one-group-at-a-time. Male and female patients are plotted in sky blue and pink colors, respectively, and each region is plotted with its own symbol type using the `pch` parameter.

Three separate figure legends were added to this complicated scatter plot figure to show multiple concepts. The first `legend()` command is identical to the command used for the bar chart in the previous example, except a `title` parameter was added to show the top right legend identifies points by gender. The second `legend()` command uses a `pch` parameter instead of the `fill` parameter, so each group can be identified by its symbol type. A researcher might choose to display both of these two figure legends, so readers could properly distinguish between the genders separated by color and the regions separated by symbol type. It is important to note that two or more legends can be applied to the same figure. But what if you want all of the information in a single legend? The third `legend()` command is redundant with the first two legends, but it demonstrates one creative way to combine both the gender and region elements into a single legend. The `fill` parameter is not used in the third legend, but a careful choice of `pch = 15` and for the "Female" and "Male" yields a filled square symbol that can be colored with the `col` parameter. The remaining region symbols are colored black for simplicity.

3.2.2 Adding text labels inside a graph

Just like the `abline()`, `points()`, `lines()` and `legend()` commands can all be used to add new features to an existing graph, there is also a `text()` command that can be used to add new text labels to specific locations within a graph. This can be useful to identify outliers on a bar chart or regression plot, to identify specific patients of interest in a medical study or just to add general comments for your readers.

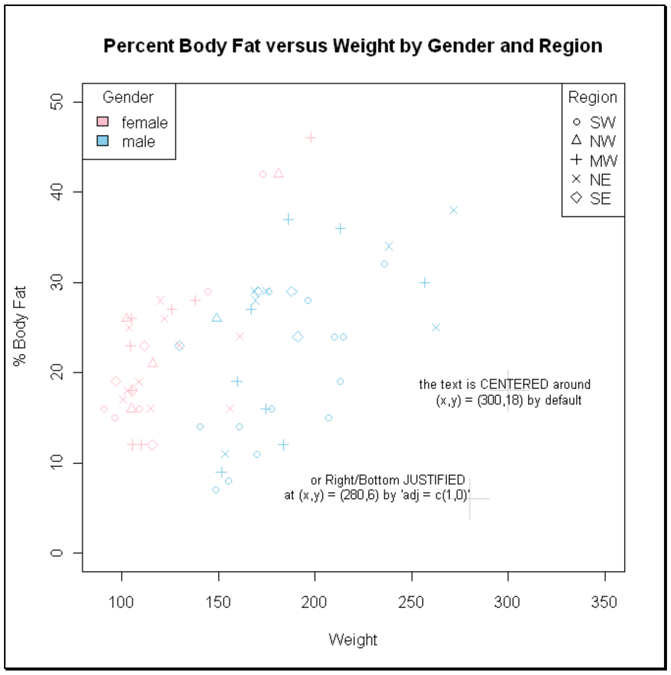


Figure 40. General comments added to a scatter plot

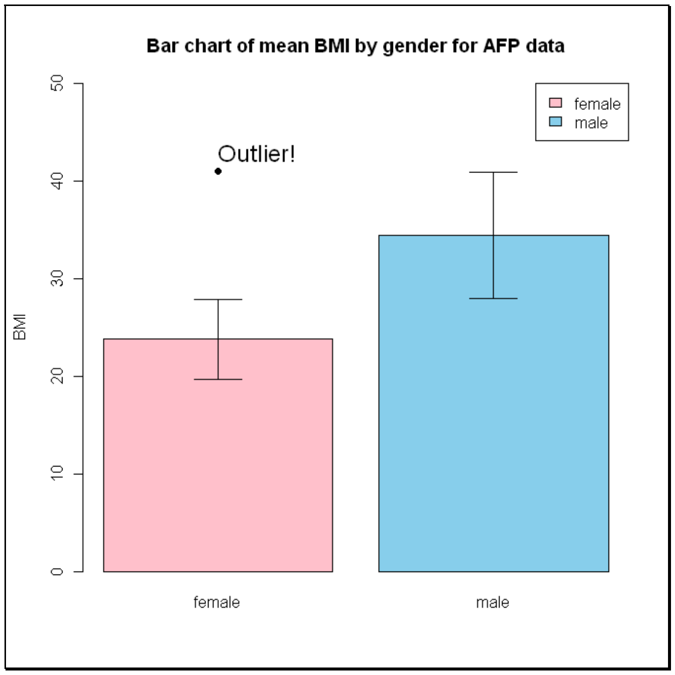


Figure 41. Outlier identified in a bar chart

```
> # Create multicolor, multisymbol scatter plot as above
> plot(AE[indx.FSW,5],AE[indx.FSW,6],xlim=c(90,350),ylim=c(0,50),main=main,
      xlab=xlab,ylab=ylab,col="pink")
> points(AE[indx.MSW,5],AE[indx.MSW,6],col="sky blue")
> points(AE[indx.FNW,5],AE[indx.FNW,6],col="pink",pch=2)
> points(AE[indx.MNW,5],AE[indx.MNW,6],col="sky blue",pch=2)
> points(AE[indx.FMW,5],AE[indx.FMW,6],col="pink",pch=3)
> points(AE[indx.MMW,5],AE[indx.MMW,6],col="sky blue",pch=3)
> points(AE[indx.FNE,5],AE[indx.FNE,6],col="pink",pch=4)
> points(AE[indx.MNE,5],AE[indx.MNE,6],col="sky blue",pch=4)
> points(AE[indx.FSE,5],AE[indx.FSE,6],col="pink",pch=5)
> points(AE[indx.MSE,5],AE[indx.MSE,6],col="sky blue",pch=5)
> legend(x="topleft",legend=c("female","male"),fill=c("pink","sky blue"),
title="Gender")
> legend(x="topright",legend=c("SW","NW","MW","NE","SE"),
      pch=1:5,title="Region")
```

```
> # Add large gray cross symbols to demonstrate centering of text
> points(c(300,280),c(18,6),pch=3,cex=4,col="light gray")
> # Add default text annotations to graph
> text(300, 18, "the text is CENTERED around \n (x,y) = (300,18)
      by default",cex=0.8)
> # Add justification adjustments to text annotation
> text(280, 6, "or Right/Bottom JUSTIFIED \n at (x,y) = (280,6)
      by 'adj = c(1,0)'",adj=c(1,0),cex=0.8)
>
> # call the barplot() command for a bar chart of BMI
> mp <- barplot(height=mean.BMI,ylab="BMI",ylim=c(0,50),
      main=main,col=colors)
> # call the arrows command to overlay error bars over barplot
> arrows(X0,Y0,X1,Y1,code=3,angle=90)
> legend(x="topright",legend=c("female","male"),
      fill=c("pink","sky blue"))
> # Add outliers to bar chart graph with points() command
> points(mp[1],41,col="black",pch=16)
> # Add text label to identify outlier point on graph
> text(mp[1],42,"Outlier!",adj=c(0,0),cex=1.5)
```

The first example uses the multi-color, multi-symbol scatter plot to show how general comments can be added to any plot (Figure 40). In this special example, two large gray “cross” symbols were added in the same location as the text comments to demonstrate how the text command uses justification. Obviously, you would want to remove these points from the final figure, but this can be a helpful trick for lining up your text. The second example shows how you might use the `points()` and `text()` commands to add annotations to outlier points on a bar chart or other graphic (Figure 41). While outliers may be plotted automatically in a box plot figure, it may be necessary to plot the outliers manually in a bar chart or other graphics.

Ch. 4. Multi-panel Layouts and Image Formats

Now that you have created some individual graphs in R, it may be necessary to organize the graphs into consistently formatted multi-panel figures and export the results for publication. You can usually copy-and-paste R graphics directly from the graphics window into popular software applications like MS PowerPoint or other software, which could be used to organize and customize the graphs. You can also save graphic images to your computer in a handful of image formats (e.g. PDF, PNG, JPG, ...) from the graphics window by clicking > **File** > **Save As...** in the MS Windows or Mac OSX R GUI. These images can also be opened in PowerPoint and other software. However, it may produce better results if you edit these figures within R. The `par()` command can be used to help organize multiple graphs into paneled figures. Additionally, R includes a variety of different graphics commands to save your graphs in popular photo, drawing and document formats (e.g. .JPG, .PDF, …). These commands can provide better control over the final look of your published figures.

4.1 Modifying Several Graphics at the Same Time

4.1.1 Using par() commands to create multi-panel figures

We have already seen that `par()` commands are shared among most graphics procedures (e.g. `main` defines main titles, `col` defines colors, `cex` defines font and symbol magnification, …). We have also seen the `par(new=TRUE)` command used in between two graphics commands to create overlay figures. The `par()` command also includes the parameters `mfcol` and `mfrow` to create multi-panel figures from multiple R graphics.

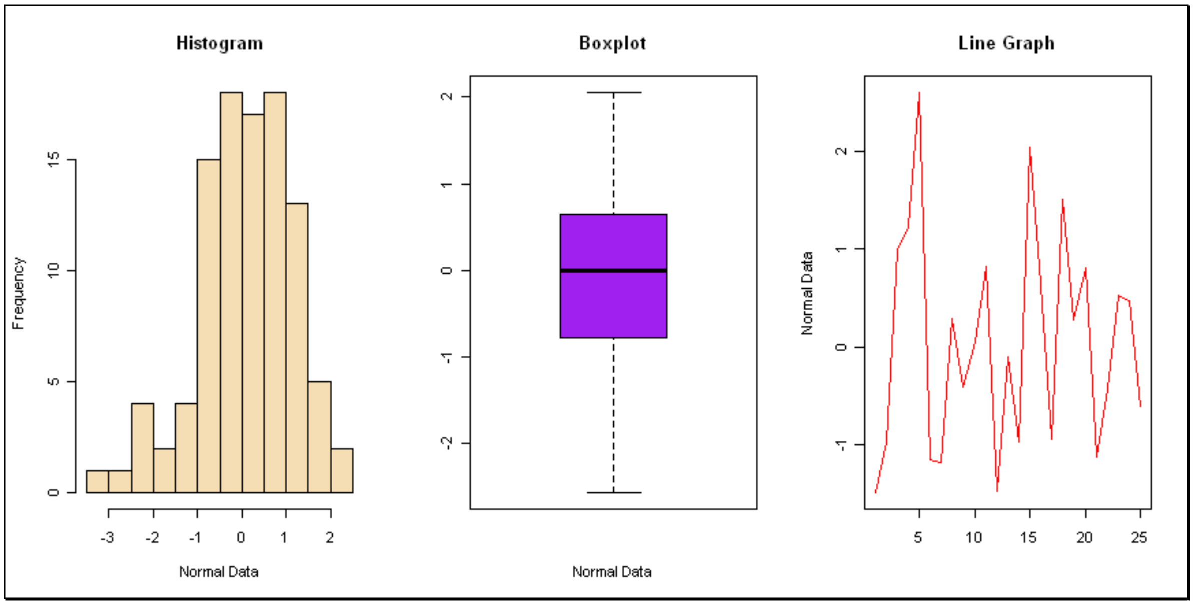


Figure 42. A multi-panel figure created with `par(mfrow=c(1,3))`

```
> # Use par(mfrow) to create a multi-panel figure
> par(mfrow = c(1,3))
> # Call three graphics commands to fill the multi-panel figure
> hist(x=rnorm(100),xlab="Normal Data",main="Histogram",
      col="wheat")
> boxplot(x=rnorm(100),xlab="Normal Data",main="Boxplot",
      col="purple")
> plot(1:25,rnorm(25),xlab="",ylab="Normal Data",col="red",
      type="l",main="Line Graph")
```

The `mfrow` parameter was used to specify that the figure should have 3 panels, arranged in one row with three columns (Figure 42). The `mfrow` parameter will fill the panels with each new graphic from left-to-right by rows, while the equivalent `mfcol` parameter will fill the panels with each new graphic from top-to-bottom by columns. Individual graphs can be assigned to specific rows and columns of the multi-panel figure using the `mfg` parameter.

4.1.2 Applying consistent graphics parameters among multiple graphs

When `par()` is used outside of several graphs, it can also be used to organize multiple graphs in a single figure, add borders and background colors, apply consistent fonts and labeling, etc (Figure 43).

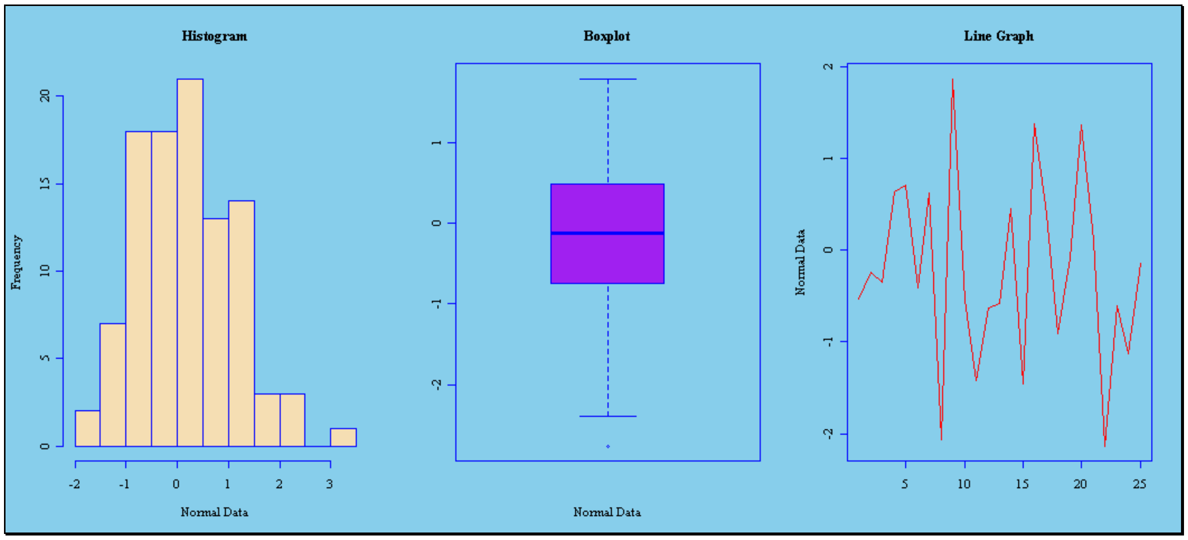


Figure 43. A paneled figure with 3 graphs with consistent parameters using par()

```
> # Use par(mfrow) to create a multi-panel figure
> par(mfrow = c(1,3), bg="sky blue",fg="blue",family="serif")
> # Call three graphics commands to fill the multi-panel figure
> hist(x=rnorm(100),xlab="Normal Data",main="Histogram",
      col="wheat")
> boxplot(x=rnorm(100),xlab="Normal Data",main="Boxplot",
      col="purple")
> plot(1:25,rnorm(25),xlab="",ylab="Normal Data",col="red",
      type="l",main="Line Graph")
```

The `par()` command is very powerful, but only two parameters were used in this simple example. The `bg` parameter is used to set the background color of the paneled figure to `"sky blue"`, the `fg` parameter sets the foreground color to `"blue"` and the `family` parameter specifies a serif font.

4.2 Exporting R Graphics

4.2.1 Exporting R graphics as .PDF and photo format output

After you create several graphics, you will want to save them for publication or for a presentation. It is possible to save your R graphics as a .PDF or .PS document using the `pdf()` and `postscript()` commands found in the `grDevices` package library. The `grDevices` package also include many popular photo image formats (e.g. .BMP, .JPEG, .PNG, .TIFF or .SVG), using the `jpeg()`, `png()`, `tiff()` or `svg()` commands. A blank image file is opened by the chosen graphic device command (e.g. the `jpeg()` command), then the open image file is filled by a plot or graphic command (e.g. the `hist()` or `par()` commands) and the image file is completed with the command `dev.off()`, which turns off the open graphic device. Most of these image format commands will have their own unique parameters to control the final size of the image, compression and other features.

```
> # Open a .JPG image to save a histogram graph
> jpeg(filename="histogram.jpeg",quality=80)
> # Plot the histogram in the open jpeg file
> hist(rnorm(100),col="wheat")
> # Turn off the open jpeg graphic device
> dev.off()
```

4.2.2 Final graphics tips

Hopefully, the previous chapters and sections have provided many useful tips that will help you customize graphs in R. Unfortunately, it is out of the scope of this manual to explore all of the specific graphic types that might be used for specific statistical tests or to visualize specific types of scientific data. However, it is relevant to end the manual with some general graphing tips that apply to all graphics and images.

If you are going to export a graphic, then think carefully about the appropriate image format. Different formats have different purposes. For example, the .PNG format is typically used for images that will be shared over the internet. The .PNG format is good for images that have large blocks of monochrome color (e.g. a bar chart or histogram) and it may be preferred over .JPG for figures with text and sharp transitions. The .TIFF format is typically the standard for publication in journals, so you may want to export to .TIFF unless instructed otherwise by the journal. The .PNG, .JPG and .TIFF formats are all raster graphics formats, which store images as a rectangular grid of pixels or colors. One disadvantage of raster graphics is that they can produce pixilated images when enlarged. An alternative to raster graphics would be a vector graphic image, like a .SVG file used in the Inkscape software system. Vector graphics are easy to edit, shrink and enlarge, but they may not produce shadings and continuous color tones as well as raster graphics.

In general, journals want your figures to be as consistent as possible, so try to keep color schemes and font choices similar among figures. Many journals will include specific instructions about image formats, file sizes, image resolution, color choices, font choices and other issues. Try to follow these recommendations as closely as possible.