METRUM INSTITUTE

MI-205: R for Pharmacometrics

Timothy T. Bergsma, Ph.D.
Senior Scientist

William Knebel, Pharm.D., Ph.D.
Principal Scientist II

James A. Rogers, Ph.D.
Principal Scientist, Statistics

Contents

Chapter 0

Getting Started

0.1 Course Introduction

MI-205: R for Pharmacometrics covers introductory through intermediate-level R programming topics with a focus on pharmacometric applications through lectures and hands-on lab sessions. Each week's topic will consist of a lecture (one hour) followed by a hands-on lab (one hour). The general plan will be as follows:

- Lectures will be on Tuesdays.
- Hands-on labs will be on Fridays (in some cases, the lecture may finish during the first part of the lab on Friday).
- Students are expected to attempt the hands-on exercises prior to the Friday lab.
- A midterm take-home exam is to be assigned at the midpoint of the course.
- A final take-home exam is to be completed by the end of the course (due one week after it is posted).
- Students will be required to complete and submit an R coding project (project details/instructions to be posted before the start of Week 4).
- Course grade will be based on the midterm (35%), final (35%), project (30%).

0.2 Metrum Institute R Web Server

All R code examples will be run on the Metrum Institute R Web Server using RStudio. Username, password, and instructions for login will be supplied via email. The login instructions

are also listed below:

1. Use a web browser (all mainstream browsers supported) and go to the web address: http://comp.metruminstitute.org:8787
2. Enter your username and password.
3. Click the **Sign In** button.

0.3 About the Course Materials

Each week, you will find a new chapter of this book along with any other course materials on the main course website (http://training.metruminstitute.org). The package will be a zip file. Normally you should:

1. Download the package to your computer.
2. Login to the RStudio web server.
3. Upload the package to your home directory on the web server using the **Upload** button in the lower right pane.
4. The zip file will be unzipped by Rstudio into the MI205 directory (this directory will be created if it does not exist).

Some book conventions to be aware of: R code that appears within a sentence will use a fixed-width font `like this`. Blocks of R code will appear as black text on a light blue background

```
like so.
```

The output of that R code may follow and will appear as blue text

```
just like this.
```

0.4 Licensing

Your use of the materials provided in this course is subject to the terms of use described in the courseware license agreement in Appendix 12.5.

Chapter 1

Basic Mechanics in R

1.1 Objectives

Afeter completing this chapter, you will be able to...

- Provide a rationale for the use of R.
- Understand how to log in and use the Metrum Institute courseware server.
- Name and define the main R objects and data types.
- Access R help.
- Create R objects and perform simple operations on them.
- Import and export simple data sets with R.
- Load an R package.
- Write and save an R script and execute code from an R script.

1.2 Introduction to R

Data manipulation, modeling, and post-processing tasks for pharmacometric applica- tions should be reproducible. They can also be complex and time consuming. The R language can perform these tasks in a way that is both reproducible and efficient. R is similar to the S language developed by AT& T Bell Labs (Lucent) and can be considered an implementation of S. It was developed as an Open Source alternative to S-PLUS and is available as free software that can be compiled and run on a variety of platforms, including *NIX (linux, NetBSD, etc.), Windows, and Mac OS X.

Downloadable binary versions for Linux, MacOSX, and Windows are available from the R project page (http://www.r-project.org).

What is R?

- Powerful, statistical, graphical, data handling and exploratory data analysis tool.
- High-level non-compiled language.
- Code is interpreted as each command is issued.
- Users can add additional functionality by defining new functions (Week 3).

1.3 R Objects and Data Types

R stores data, graphs, functions, etc. as R data structures or objects, which reside in memory and can't be read externally to R unless specifically written to a file. This is different from S-Plus, which uses available hard disk space for object storage. This has implications when reading very large (greater than available memory) files.

Some examples of R data objects are vector, matrix, list, data.frame.

R objects differ in the data types stored and the associated attributes. Some typical data types are:

- `numeric` - numbers or integers
- `character` - letters or combinations of letters and numbers
- `factor` - representation of underlying data via levels/categories
- `logical` - vector containing `TRUE`, `FALSE`, or `NA`

In dealing with R objects, we can think of moving along a complexity scale that goes from simplest (vector) to the most complex (list of data frames). As we move from simple to complex, the information the R object carries increases.

Vector

- Ordered set of values
- Indexed by `[row]` or `[col]` numbers
- Can be created with `vector()` function

Table 1.1: Vector Object

Attribute	Description
length	number of values
mode	kind of values (data type)
names	value labels

Now we can create a simple vector called 'a' using the following R code.

Listing 1.1:

```
# create a vector called _a_ containing the numbers 1 - 10
a <- 1:10
```

Listing 1.2:

```
# In the above example 'a' was 'assigned' the numbers 1 - 10
```

```
# the 'assignment' operator is '<-'

# create vector then add information
a <- vector("numeric",10)
a[1:10] <- 1:10
```

The second example may seem a non-direct route to the same results but there are times when creating an empty vector and then filling it is easier than creating the vector in one step (more on the second approach in later weeks).

Matrix

- Values arranged by rows and columns in a rectangular table
- Indexed by `[row,col]` numbers
- Can be created with `matrix()` function

Table 1.2: Matrix Object

Attribute	Description
length	number of values
mode	kind of values (data type)
dim	number of rows and columns
dimnames	row and column names

We can create a matrix called `ma` using the following R code.

Listing 1.3:

```
# create a 2x2 matrix called 'ma' containing the numbers 1 - 4
ma <- matrix(c(1:4),
             nrow = 2,
             ncol = 2,
             byrow = F
             )
ma
```

```
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

Listing 1.4:

```
# a slightly different version of 'ma' when 'byrow=T'
ma <- matrix(c(1:4),
             nrow = 2,
             ncol = 2,
```

```
              byrow = T
              )
ma
```

```
     [,1] [,2]
[1,]    1    2
[2,]    3    4
```

Listing 1.5:

```
# create a 2x2 matrix containing characters rather than numbers
# we do not specify 'byrow' so the default value of 'F' is used
clra <- c("red","white")
clrb <- c("blue","stars")
flag <- matrix(c(clra,clrb),
               nrow = 2,
               ncol = 2
               )
flag
```

```
     [,1]    [,2]
[1,] "red"   "blue"
[2,] "white" "stars"
```

Listing 1.6:

```
# can't mix data types in a matrix and get expected results
a <- 1:2
mixm <- matrix(c(clra,a),
               nrow = 2,
               ncol = 2
               )
mixm
```

```
     [,1]    [,2]
[1,] "red"   "1"
[2,] "white" "2"
```

Notice all elements are `character`, as denoted by `" "` around elements.

`matrix()` will return a character matrix if there are any non-numeric columns.

List

- An ordered collection of objects know as components.
- Allows combination of components of different data types and lengths.
- Indexed by component number .

- Can be created with the `list()` function.
- Each component in a list can be indexed by a number or by $.

Listing 1.7:

```
b[[1]]
b$name
```

It important to distinguish between `b[[1]]` and `b[1]`. `[[...]]` is the operator used to select a single element and `[...]` is a general subetting operator (more on subsetting later).

Table 1.3: List Object

Attribute	**Description**
length	number of components
mode	all lists are of mode `list`
names	name of each component

We can create a list called `Lsta` using the following R code.

Listing 1.8:

```
# create a list with the components 'breed', 'ages', 'colors'
# we can start by creating each component then making the list
breed <- c("beagle","irish setter", "mix")
ages <- c(1, 5, 3)
colors <- c("brown","tan","red")
Lsta <- list(breed, ages, colors)
Lsta
```

```
[[1]]
[1] "beagle"       "irish setter" "mix"

[[2]]
[1] 1 5 3

[[3]]
[1] "brown" "tan"   "red"
```

Listing 1.9:

```
# we could have also created 'Lsta' in one step
Lsta <- list(breed = c("beagle","irish setter", "mix"),
             ages = c(1, 5, 3),
             colors = c("brown","tan","red")
             )
```

Notice that `Lsta` contains character and numeric components and the original `type` of each component is preserved.

Data Frame

- Spreadsheet-type format, similar to `matrix`, that can be created with `data.frame()` function.
- Allows combination of variables (columns) of different data types.
- Indexed by `[row,col]` numbers.
- Default data object for `read.table()` function and its variants (`read.csv()`, `read.delim()`).
- Each column is a vector and is indicated by $.

e.g. data frame: `data1`
DV column: `data1$DV`

Table 1.4: Data Frame Object

Attribute	Description
length	number of variables in the data frame
mode	all data frames are of mode list
names	names of the varibles (columns)
row.names	names of the rows in the data frame
dim	number of rows and columns as [r,c]
dimnames	row and column names

We can create a data frame called `df1` using the following R code.

Listing 1.10:

```
# create a data frame called 'df1' from five vectors
id <- c(1,3,5,7)
age <- c(35, 46, 50, 25)
wt <- c(70, 100, 67, 40)
sex <- as.factor(c("male","male","female","male")
                 )
trt <- as.logical(c('T','F','T',NA)
                  )
df1 <- data.frame(id, age, wt, sex, trt)
# typing the data frame name (df1) at the R prompt
#  prints out the components of df1

df1
```

```
  id age  wt    sex   trt
1  1  35  70   male  TRUE
2  3  46 100   male FALSE
3  5  50  67 female  TRUE
4  7  25  40   male    NA
```

The data type of vectors combined to make a data frame are preserved. This can be proven by using the `str()` function.

Listing 1.11:

```
str(df1)
```

```
'data.frame':   4 obs. of  5 variables:
 $ id : num  1 3 5 7
 $ age: num  35 46 50 25
 $ wt : num  70 100 67 40
 $ sex: Factor w/ 2 levels "female","male": 2 2 1 2
 $ trt: logi   TRUE FALSE  TRUE    NA
```

`str()` indicates the R object `df1` is a data frame with 4 observations and 5 variables with the first three variables being `numeric` and the next two being a `factor` and `logical`, respectively.

1.4 Getting Help Inside and Outside of R

R has a very extensive help that can be accessed in a variety of ways. The R help information has been provided via a link in the RStudio interface.

Lets take a quick tour of the R help browser. Log onto the web server and click the **Help** button.

Other ways of getting help with R:

- Online Manuals - under Help Menu
- `help(function name)` or `?function name` from the R prompt
- Venables & Ripley. Modern Applied Statistics with S (4th edition). 2002
- Paul Murrell. R graphics. 2005.
- R-project web page http://www.r-project.org and mailing lists: http://www.r-project.org/mail.html

1.5 Create R Objects and Perform Simple Operations

Now that we understand the basic building blocks of R, we can begin to operate on objects we have created.

Some examples for creating simple R objects

- create a simple vector of numbers and 'assign' it to a variable:

  Listing 1.12:

  ```
  a <- c(1,2,3,6,10)
  ```

  The values 1,2,3,6, and 10 were assigned to `a` using `<-`.

- use the colon (:) operator or `seq()` to generate a sequence of numbers:

  Listing 1.13:

  ```
  b <- c(1:5)
  b1 <- seq(from=1,to=5)
  ```

  `b` and `b1` contain the same set of values.

- `seq()` can also be used to create more complicated sequences

  Listing 1.14:

  ```
  b2 <- seq(from=1, to=5, by=0.5)
  b2
  ```

```
[1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

Listing 1.15:

```
b3 <- seq(from=1, to=5, length.out=10)
b3
```

```
 [1] 1.000000 1.444444 1.888889 2.333333 2.777778 3.222222
 [7] 3.666667 4.111111 4.555556 5.000000
```

- `rep()` can be used to replicate an object.

Listing 1.16:

```
d <- c(3,4)
d1 <- rep(d, times=4)
d1
```

```
[1] 3 4 3 4 3 4 3 4
```

`d1` was created by replicating `d` four times.

- could also use a combination of the above:

Listing 1.17:

```
e <- rep(seq(from=1, to=10), times=4)
```

`e` will contain the numbers 1 - 10 repeated four times.

Vector operations

- Vectors can be used in arithmetic expressions with the evaluation performed element by element

Listing 1.18:

```
a <- c(1:5)
b <- c(2:6)
a+b
```

```
[1]  3  5  7  9 11
```

- Vectors can be of differing length; if so, they are recycled to complete the operation.

Listing 1.19:

```
a <- c(1:5)
b <- 2
c <- 1
a*b+c
```

```
[1]  3  5  7  9 11
```

- Available elementary operators are +, −, ⋆ ,/ , and ˆ (raising to a power)
- Also available are the common arithmetic functions `log`, `exp`, `sin`, `cos`, etc.
- Standard `summary` functions are also available `min()`, `max()`, `mean()`, etc. as well as `summary()` that incorporates a number of the standard summary functions.

Data frame operations

- Operations on data frames can be performed in a similar fashion to vectors
- The operations can be performed on a column-by-column basis, over the entire data frame, or over a portion of the data frame.
- Operating on a column(s) of a data frame requires the user to indentify the column(s) to use for the operation.
  - identify/see the `conc` column in the `a` data frame using `a$conc`
  - Operate on that column using a similar notation:

Listing 1.20:

```
a <- data.frame("time" = c(0:10),
                "conc" = c(0, 20, 100, 96, 94, 92,
                                  90, 60, 30, 5, 1),
                "dose" = rep(5,11)
                )
a$conc <- a$conc * 10
# we have multiplied the "conc" column by 10 and put the
# new values back into the 'a' data frame
a$conc10 <- a$conc * 10
```

This may be a better way to do it because you can keep track of what was done.

  - Can also multiply a data frame by a scalar (single value) or vector (set of values).

Listing 1.21:

```
a <- data.frame("time" = c(0:10),
                "conc" = c(0, 20, 100, 96, 94, 92,
                                  90, 60, 30, 5, 1),
                "dose" = rep(5,11)
                )
a2 <- a*5
# we have multiplied all of the columns in 'a' by 5.
# BE CAREFUL this only works if there
# are no character or factor columns
```

```
mly <- c(1,2)
a3 <-a*mly
# This code works but lets see what the results are.
a3
```

```
   time conc dose
1     0    0    5
2     2   20   10
3     2  200    5
4     6   96   10
5     4  188    5
6    10   92   10
7     6  180    5
8    14   60   10
9     8   60    5
10   18    5   10
11   10    2    5
```

R 'recycled' the `mly` vector for the multiplication because it was shorter in length than the data frame.

Subsetting a data frame

- Use the subset (`[]`) operator to select some or many columns or rows
- Very powerful and often used operator in R.
- Rows and columns are specified as `[row, column]`.
- Some examples:
  - select a given column

Listing 1.22:

```
a <- data.frame("time" = c(0:10),
                "conc" = c(0, 20, 100, 96, 94, 92,
                                  90, 60, 30, 5, 1),
                "dose" = rep(5,11)
                )
# data frame 'a' contains three columns
# with 11 rows per column
# select only the 'time' and 'conc' columns and
# place in new data frame call 'f1'
f1 <- a[ ,c("time","conc")]
#or
f1 <- a[ ,c(1,2)]
```

Why might the first approach be better/safer?

– Select a row or set of rows:

Listing 1.23:

```
a <- data.frame("time" = c(0:10),
                "conc" = c(0, 20, 100, 96, 94, 92,
                                    90, 60, 30, 5, 1),
                "dose" = rep(5,11)
                )
# select only the first 5 rows and all columns
f2 <- a[1:5,]
# select a$time <=5
f3 <- a[a$time<=5,]
```

Go to Rstudio web interface, navigate to the basicmechanics directory under MI205, and click the on the basicmechanics.R file. This should open the `basicmechanics.R` script in its own window. Use the script to create and manipulate R objects.

1.6 Import and Export Simple Data Sets with R

R has the ability to import a variety of data set types:

- ASCII text with any delimiter (preferred).
- Excel: save as *.csv.
- SAS transport (*.xpt) file import and export using SASxport package.
- Additional statistical and database file formats, including Minitab, S-PLUS, SPSS, Stata.

The main function for importing data is `read.table()`.

Listing 1.24:

```
read.table(file, header=FALSE, sep, row.names, col.names,
                    as.is=F, na.strings="NA", skip=0)
# importing a simple comma delimited *.csv might look like

test <- read.table(file="/home/billk/example.csv",
                   header=TRUE,
                   sep=".",
                   skip=0)
```

R makes an attempt to identify the data type (`numeric, character, factor`) of each column when `read.table()` is used.

- Default behavior is to convert character variables to factors.
- Variable `as.is` controls this conversion, by default it is set to `as.is=FALSE`.
- Setting `as.is=TRUE` suppresses conversion of characters to factors (in many cases it is easier to set `as.is=T` and perform conversion after data has been imported).

There are also variants of `read.table()` specifically for reading in *.csv files `read.csv()` and delimited files `read.delim()`.

The main function for exporting files from R is `write.table()` .

Listing 1.25:

```
write.table(data, file = "", sep = ",", append = F, quote = F,
            col.names=T, row.names=F, na = 'NA', eol = "\n")
# exporting a simple R data frame might look like

write.table(x = test,
            file = "/home/billk/exampleout.csv",
            sep = ",",
            append = F,
            quote = F,
            col.names=T,
            row.names=F,
            na = '.')
```

This will write out the 'test' R object to 'exampleout.csv' using the PATH defined in 'file'.

There is also a variant of `write.table()` specifically for writing *.csv files `write.csv()`.

SAS transport files (specifically *.xpt) can be written using the SASxport package (more on this in later weeks).

Go to RStudio web interface, navigate to the basicmechanics directory under MI205, and click the on the ImportExport.R file. This should open the ImportExport.R R script in its own window. Review importing and exporting of files.

1.7 Load an R package

The `library()` function is used to load R packages when R is running. The syntax for this command is:

Listing 1.26:

```
library("package name")
```

Listing 1.27:

```
library(lattice) # loads the lattice graphics package
installed.packages() # lists all available packages for a given R
   installation
```

On RStudio the available packages can be found by looking in the **Help** link and then clicking on the **Packages** link.

If a package is not available for a given R installation, it must be installed from the R gui then loaded with library(PackageName) from within R. All packages for the MI205 course have been pre-installed on the web interface and can be loaded using the `library()` function.

1.8 Write and Save an R Script and Execute Code from an R Script

Previous sections have demonstrated running R code from a script (`basicmechanics.R`). The RStudio interface can be used to create an R script, add code to it, execute all or portions of it, and save the file.

1.9 Homework Problems

1. Import the `week1.csv` data file and determine the number of rows and columns in the imported data set.
2. Summarize the `time`, `dose`, and `conc` columns. What are the minimum, maximum, mean and median of each column?
3. Create a new data frame that only contains the `id`, `time`, and `conc` columns.
4. In the subsetted data frame, create a new column called `nid` that is an exact copy of `id`.
5. Export the data frame in 4 as a csv file named `week1new.csv`.

Chapter 2

Basic Plotting

2.1 Objectives

After completing this chapter, you will be able to ...

- Provide a high-level description of the two basic graphics paradigms in R and their relationship to trellis/lattice graphics in R (and S+).
- Make a rudimentary scatterplot with traditional graphics.
- Make a rudimentary scatterplot with lattice graphics.
- Open and close devices, including saving a plot to a file (e.g. pdf, png, bmp)
- Make trellised and non-trellised scatterplots in lattice, using the following arguments:
  - the `main`, `xlab`, and `ylab` arguments
  - the `type` argument
  - the `subset` argument
  - the `groups` argument
  - the `auto.key` argument
  - the `col`, `pch`, `lty` and related arguments
  - the `relation` component of the `scales` argument
- Control the panel dimensions (layout) for trellised scatterplots.
- Control the order of panels within a trellised scatterplot.
- Toggle between the two basic types of panel "strips".

2.2 Introduction

There are two basic graphics paradigms in R: traditional graphics and grid graphics [1]. Examples of packages and functions from these two paradigms are shown in Figure 2.2. In general, the two paradigms are non-intersecting: what works in traditional graphics will not work in grid graphics, and vice versa.

In this course we will focus primarily on the `lattice` package, written and maintained by Deepayan Sarkar [2]. The `lattice` package makes it very easy to produce multi-panel plots (such as have been popularized by Spotfire visualization software), but it is also a great environment for making a wide variety of single-panel plots.

Since we will be spending so much time with the `lattice` package, we should note that this is one bit of your R education that won't translate to S+. At first glance, the `lattice` package appears to be similar to its S+ analog, `trellis`, to the extent that they produce similar types of displays and the high-level syntax is more or less the same. This is a superficial similarity — `lattice` code of even moderate complexity generally will not work in `trellis`. I will point out a few non-translatable bits of code as we go along. For more on this, see the main help page for `lattice`.

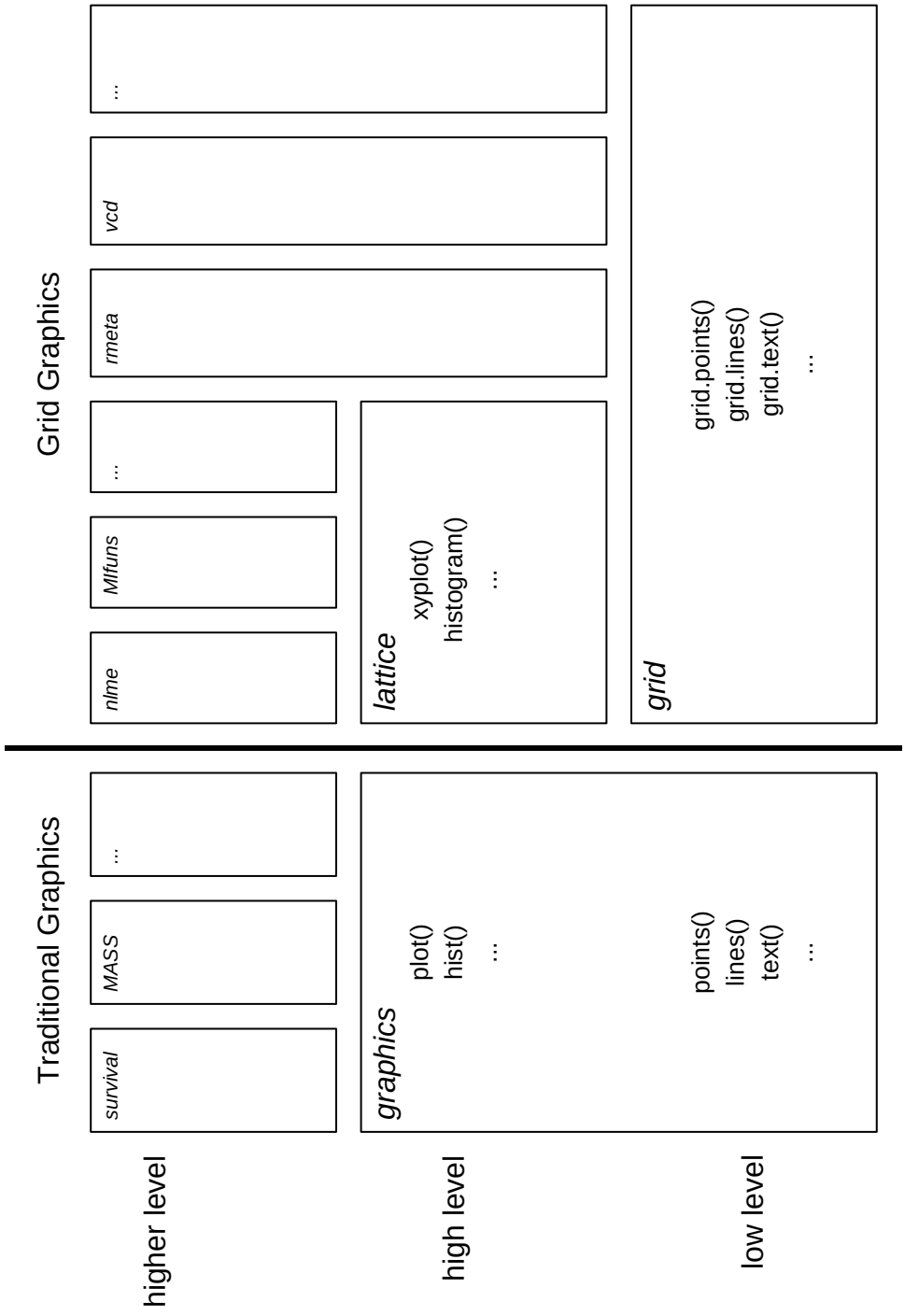


Figure 2.1: Examples of packages (in italics) and functions (followed by parentheses) from both traditional and grid graphics paradigms.

Throughout this lesson we will work with a single data set. It's a fake data set, but it's moderately realistic and presents you with a number of challenges that are typical to pharmacometric data sets. Create this data set by executing the code below. Based on what you learned in the first lession, you probably *could* figure out what most of this code is doing, but that's not the point of this lesson, so let's just accept this code as a black box for now so we can get right to the plotting.

Listing 2.1:

```
set.seed(1)
nSubj <- 40
doses <- c(0, 10, 30, 100)
times <- seq(0, 9, 3)
dat <- expand.grid(Subject = 1:nSubj,
                   Month = times)
dat <- as.data.frame(dat)
dat$Dose <- doses[as.numeric(dat$Subject) %% length(doses) + 1]
dat$Exposure <- dat$Dose * exp(rnorm(nrow(dat), 0, 0.3))
noise <- rep(rnorm(nSubj, 0, 0.2), length(doses)) + rnorm(nrow(dat
   ), 0, 0.2)
emax <- c(0, 50, 100, 100)[match(dat$Month, times)]
dat$Response <- with(dat, 10 + emax * Exposure / (Exposure + 25))
   * exp(noise)
dat$Response <- pmin(dat$Response, 100)
```

Let's take a quick look at what we just created, by sampling a few rows from this data set:

Listing 2.2:

```
sample.rows <- sample(1:nrow(dat), 10)
dat[sample.rows, ]
```

```
    Subject Month Dose  Exposure   Response
1         1     0   10  8.286676  11.816366
96       16     6    0  0.000000  10.201445
144      24     9    0  0.000000   7.037375
111      31     6  100 82.636316 100.000000
41        1     3   10  9.518412  29.833741
132      12     9    0  0.000000   7.462788
52       12     3    0  0.000000  11.080414
89        9     6   10 11.174012  43.325292
66       26     3   30 31.748170  38.827614
8         8     0    0  0.000000   8.380869
```

As you can see, we have longitudinal observations. Importantly, this longitudinal data set is organized in the "tall-skinny" format with the time variable as one of the columns, rather than the "short-fat" format where each time point is represented in a different column. Recent versions of `lattice` can handle "short-fat" data to some extent, but in general a "tall skinny" data set is a better starting point. Later in the course you will learn how to reshape data sets from one format to the other within R.

2.3 Graphics Devices

When you work directly with a standard, locally-installed version of R, or when you work on RStudio, you can make calls to graphics functions and the result will just magically appear on the screen. Graphics "devices" are being used to present the graphics on your display. This is often taken care of for you behind the scenes, but it doesn't hurt to understand the basics.

To determine the device that RStudio has turned on for us, we can do:

Listing 2.3:

```
dev.cur()
```

This screen device is fine for our current purposes, but sometimes we want to be able to write our plots to files in a scripted manner. For that purpose we could turn on a pdf device:

Listing 2.4:

```
pdf(file = 'test.pdf')
```

From now until we turn that device off, all the graphics that we create will be directed to the file `test.pdf`. If we should forget what the current device is, we can always checks again with:

Listing 2.5:

```
dev.cur()
```

While leaving the pdf device active, let's plot `Exposure` versus `Response`:

Listing 2.6:

```
plot(dat$Exposure, dat$Response)
```

and maybe we want to add a horizontal reference line at $y = 50$:

Listing 2.7:

```
abline(h = 50)
```

Now, go and take a look at `test.pdf`, and you should see ... nothing (or maybe a message that the file is corrupt). That's because we haven't told the pdf device that we are done sending content to it. For all of the graphics devices that write to files, we turn the device off in order to finish writing the file and get something in valid format:

Listing 2.8:

```
dev.off()
```

A valid pdf file now awaits us.

Since we will be opening up and closing pdf and png devices frequently, you may want to just invoke `pdf()` and `png()` without any arguments. A default file name will be used.

Listing 2.9:

```
pdf()
plot(1:10, 11:20)
dev.off()
```

In this case the file created is `Rplots.pdf`.

NB: From here on out, it will be understood that every set of plotting instructions must be preceded by the opening of a device and followed by the closing of that device. We won't continue to show this explicitly in the code examples.

2.4 Traditional Graphics

We have already seen one way of using the traditional graphics function `plot()` to produce a scatterplot. That used what is known as the "default method". It worked fine in that case,

but it can get cumbersome: suppose we don't want to include the placebo group in our plot (maybe because we want to plot versus log exposure). To accomplish this with the default method we could do:

Listing 2.10:

```
plot(log(dat$Exposure[dat$Dose != 0]), dat$Response[dat$Dose !=
   0])
```

Still not too bad, but do we really have to specify the data set four times, and the subset two times? It is often more convenient to use the "formula method". One way to use the formula method would be:

Listing 2.11:

```
plot(dat$Response[dat$Dose != 0] ~ log(dat$Exposure[dat$Dose !=
   0]))
```

but this doesn't gain us anything in terms of convenience. A better way to use the formula method allows us to specify the data set only once, and to specify the subset only once:

Listing 2.12:

```
plot(Response ~ log(Exposure),
     data = dat,
     subset = Dose != 0)
```

Maybe we also want to add a title, and some more informative axis labels:

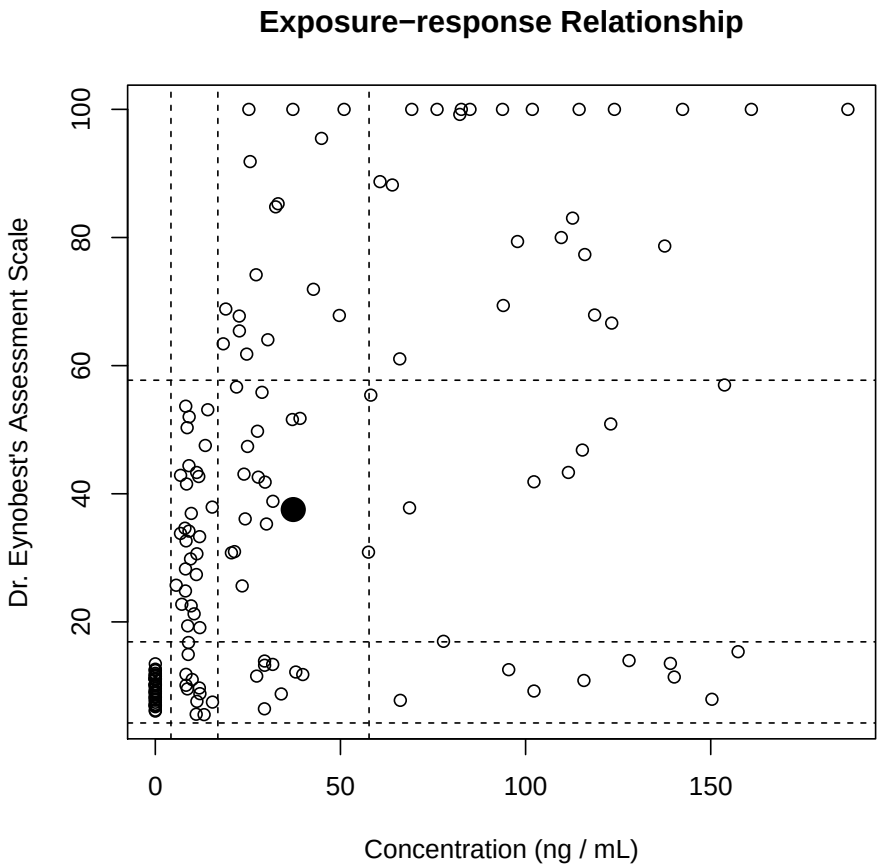
Listing 2.13:

```
plot(Response ~ Exposure,
     data = dat,
     main = 'Exposure-response Relationship',
     xlab = 'Concentration (ng / mL)',
     ylab = "Dr. Eynobest's Assessment Scale"
     )
```

Note that `plot()`, in addition to actually plotting the points, has been drawing the axes with tick marks and adding labels for each axis. It can do a lot of other things too, like adding a title. All those extras are great the first time around, but when we add to an existing plot, we need lower levels functions like `points()` and `abline()`:

Listing 2.14:

```
plot(Response ~ Exposure,
     data = dat,
     main = 'Exposure-response Relationship',
     xlab = 'Concentration (ng / mL)',
     ylab = "Dr. Eynobest's Assessment Scale"
     )
points(mean(dat$Exposure), mean(dat$Response),
       pch = 19, # this specifies solid circles
       cex = 2) # this specifies 2x the default size
v.reflines <- quantile(dat$Exposure, probs = c(0.25, 0.5, 0.75))
h.reflines <- quantile(dat$Response, probs = c(0.25, 0.5, 0.75))
abline(v = v.reflines, lty = 2) # lty = 2 means dashed lines
abline(h = v.reflines, lty = 2)
```



2.5 Lattice Graphics

Unlike the `graphics` package that we used to create our traditional graphics, `lattice` is not loaded by default. We need to load it with the `library()` command:

Listing 2.15:

```
library(lattice)
```

At a high level, the `lattice` function `xyplot()` works similarly to the formula method of the traditional `plot()` function. In it's simplest usage, it just takes a single formula that references two vectors:

Listing 2.16:

```
xyplot(1:10 ~ 11:20)
```

Let's reproduce one of our examples from traditional graphics:

Listing 2.17:

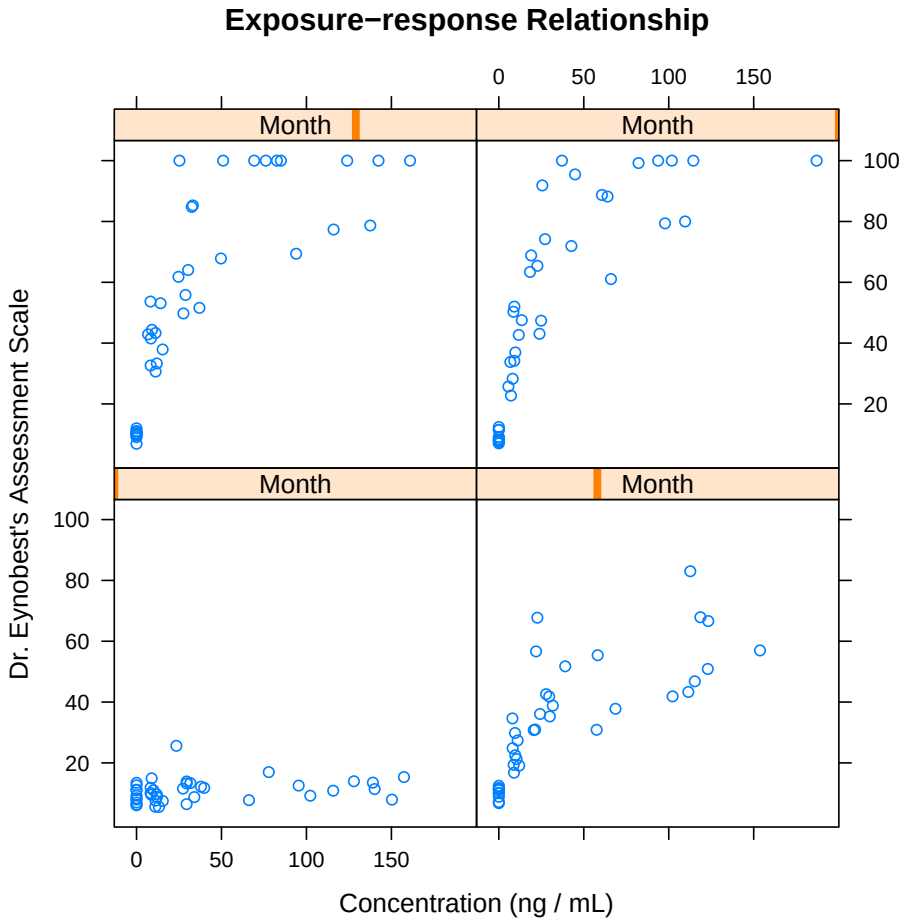
```
print(
      xyplot(Response ~ Exposure,
             data = dat,
             main = 'Exposure-response Relationship',
             xlab = 'Concentration (ng / mL)',
             ylab = "Dr. Eynobest's Assessment Scale"
             )
      )
```

Using a conditioning variable: trellising

At this point, you may be wondering what `xyplot()` has brought to the party other than some differently colored dots. Here's your answer:

Listing 2.18:

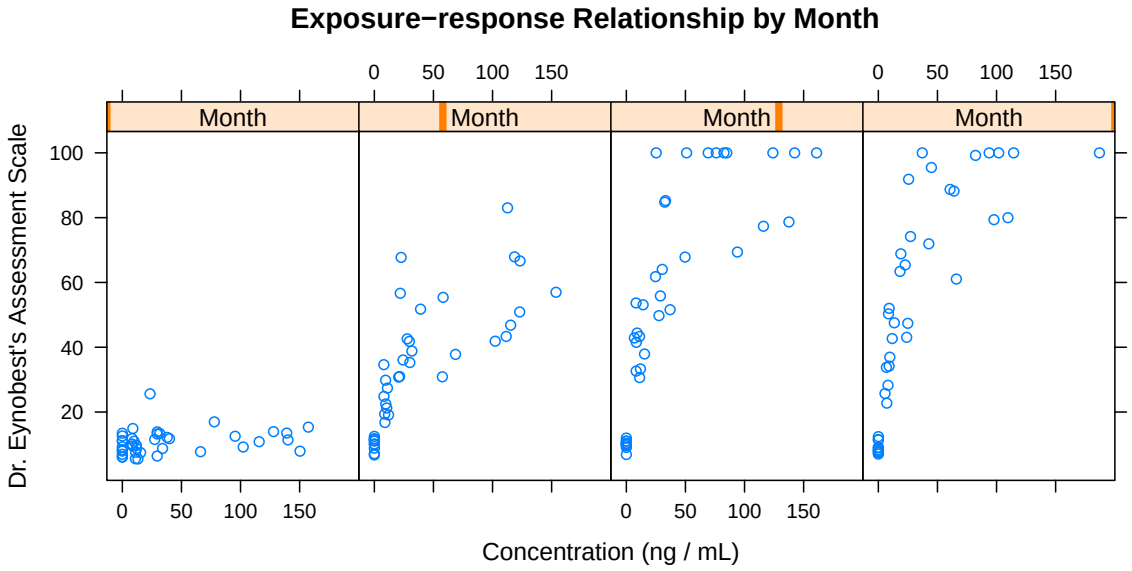
```
print(
      xyplot(Response ~ Exposure | Month,
             data = dat,
             main = 'Exposure-response Relationship',
             xlab = 'Concentration (ng / mL)',
             ylab = "Dr. Eynobest's Assessment Scale"
             )
      )
```



That default arrangement of the panels makes it a bit hard to compare responses between months. A more informative arrangement would be to put all four panels in a single row. We do this with the `layout` argument:

Listing 2.19:

```
print(
      xyplot(Response ~ Exposure | Month,
             data = dat,
             layout = c(4, 1), # this means 4 columns, 1 row
             main = 'Exposure-response Relationship by Month',
             xlab = 'Concentration (ng / mL)',
             ylab = "Dr. Eynobest's Assessment Scale"
             )
      )
```



The "strips" on each panel have a little dark band, indicating where that panel's `Month` value falls relative to the other `Month` values (note the within-strip position of the dark band progresses from left to right as the panels progress from left to right). Lattice is presenting the information this way because it sees that `dat$Month` is a numeric variable.

Listing 2.20:

```
str(dat$Month)
```

```
 num [1:160] 0 0 0 0 0 0 0 0 0 0 ...
```

A different style of strip will be used if we change `dat$Month` to character:

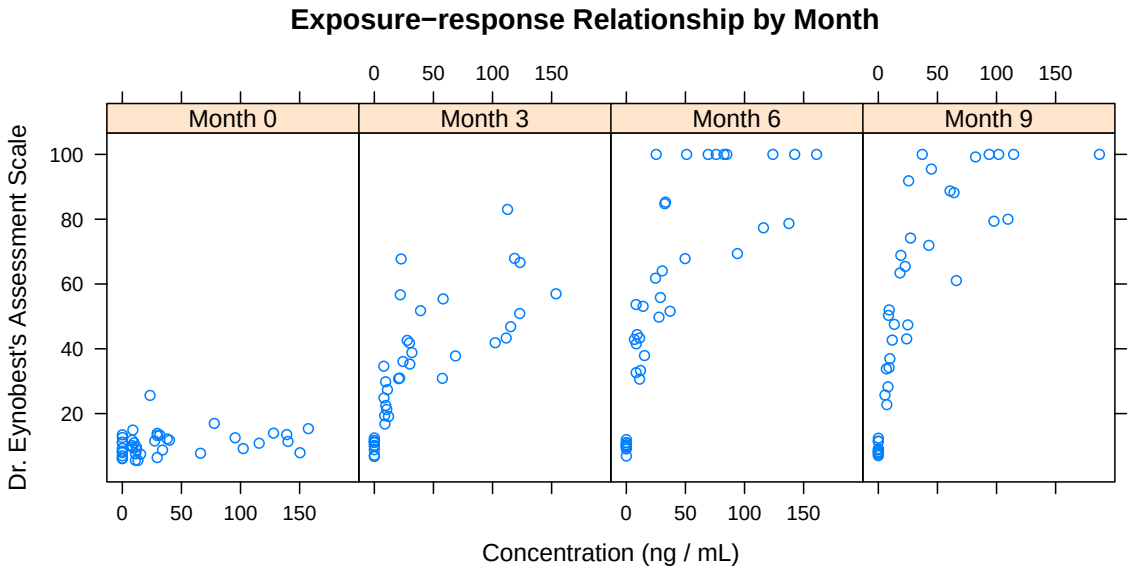
Listing 2.21:

```
dat$Month2 <- paste("Month", dat$Month)
str(dat$Month2)
print(
      xyplot(Response ~ Exposure | Month2,
             data = dat,
             layout = c(4, 1),
             main = 'Exposure-response Relationship by Month',
             xlab = 'Concentration (ng / mL)',
             ylab = "Dr. Eynobest's Assessment Scale"
             )
      )
```

This is a fairly common thing to want to do, so be aware that you can do this conversion on the fly (this will save you from creating a lot of extra variables):

Listing 2.22:

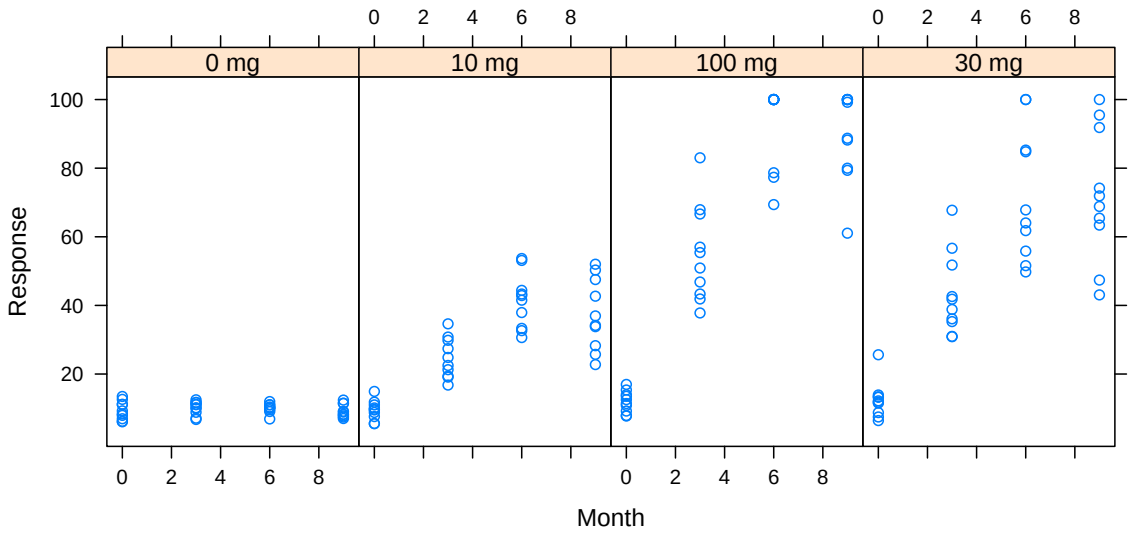
```
print(
      xyplot(Response ~ Exposure | paste("Month", Month),
             data = dat,
             layout = c(4, 1),
             main = 'Exposure-response Relationship by Month',
             xlab = 'Concentration (ng / mL)',
             ylab = "Dr. Eynobest's Assessment Scale"
             )
      )
```



Let's try a similar trick using `Dose` as the conditioning variable:

Listing 2.23:

```
print(
      xyplot(Response ~ Month | paste(Dose, 'mg'),
             data = dat,
             layout = c(4, 1),
             )
      )
```



Now, unless you really love to break conventions, you probably would prefer to have the value of `Dose` increase as we go from left to right. There is an easy way to do this, and a hard way. We are going to do the hard way, because it's a chance to learn a little bit about factors.

First let's make a character version of Dose:

Listing 2.24:

```
dat$Dose2 <- paste(dat$Dose, 'mg')
```

While we're at it, let's relabel '0 mg' as 'Placebo':

Listing 2.25:

```
dat$Dose2[dat$Dose2 == '0 mg'] <- 'Placebo'
```

Now let's convert it from a character vector to a factor. This is exactly what lattice does behind the scenes.

Listing 2.26:

```
dat$Dose2 <- factor(dat$Dose2)
```

By default, factor levels are sorted on the underlying data type (in this case character, so they are sorted lexically):

Listing 2.27:

```
levels(dat$Dose2)
```

```
[1] "10 mg"   "100 mg"  "30 mg"   "Placebo"
```

We can use what we know about subscripting to permute the factor levels:

Listing 2.28:

```
levels(dat$Dose2)[c(4, 1, 3, 2)]
```

```
[1] "Placebo" "10 mg"   "30 mg"   "100 mg"
```

Now let's assign this permuted vector to be the levels for `dat$Dose2`:

Listing 2.29:

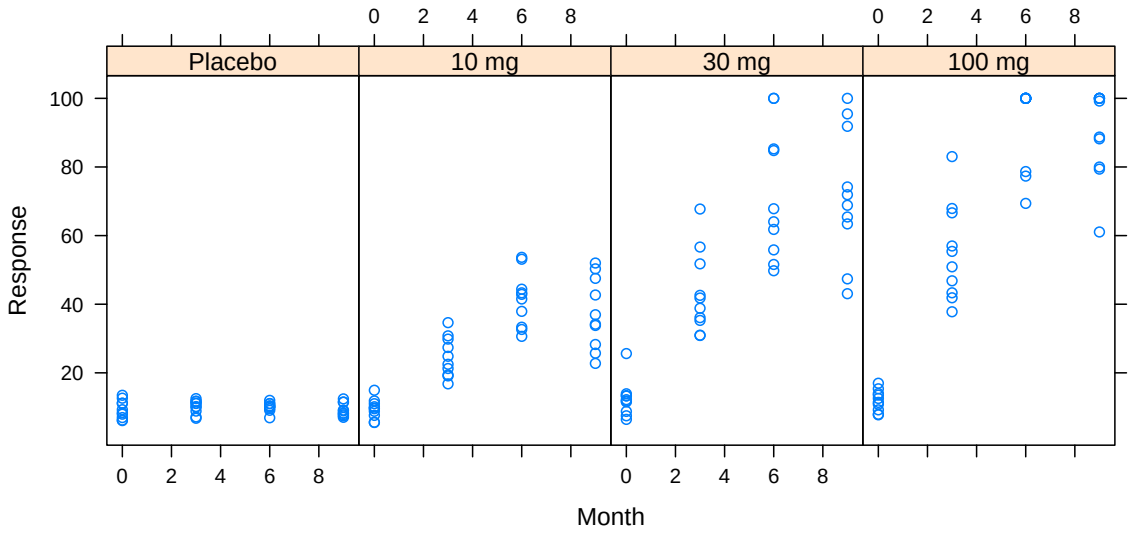
```
newlevs <- levels(dat$Dose2)[c(4, 1, 3, 2)]
dat$Dose2 <- factor(dat$Dose2, levels = newlevs)
levels(dat$Dose2)
```

```
[1] "Placebo" "10 mg"   "30 mg"   "100 mg"
```

If you find yourself doing this sort of thing a lot, you should take a look at the `mixedsort()` function in package `gtools` and the `reorder()` function in package `gdata`. Those funcions provide the sort of hybrid lexical / numeric sorting that one typically wants for dose labels.

Listing 2.30:

```
print(
      xyplot(Response ~ Month | Dose2,
             data = dat,
             layout = c(4, 1),
             )
      )
```



Now that you've been through that pain, here's the easy way we could have done it, using the original `Dose` variable:

Listing 2.31:

```
print(
      xyplot(Response ~ Month | paste(Dose, 'mg'),
             data = dat,
             index.cond = list(c(1, 2, 4, 3))
      )
      )
```

Using the groups argument and legends/keys

So far our plots have been blind to the fact that each subject has multiple associated data points. We can distinguish points from different subjects using the `groups` argument:

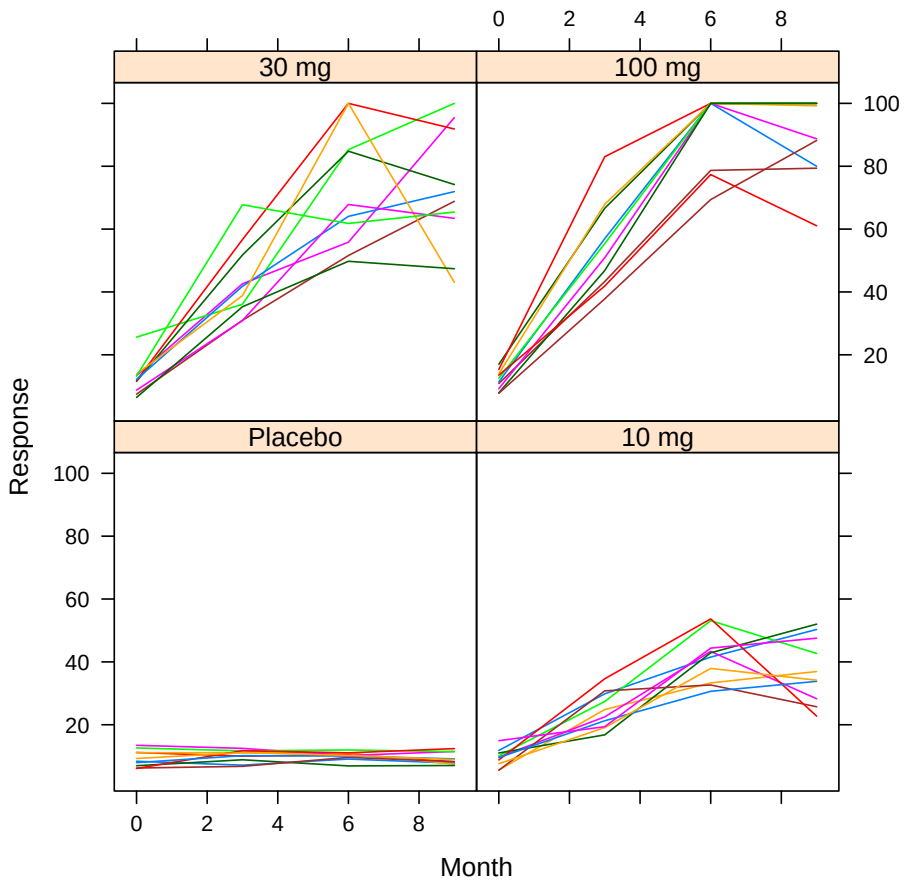
Listing 2.32:

```
print(
      xyplot(Response ~ Month | Dose2,
             groups = Subject,
             data = dat)
      )
```

And let's link each patient's points with line segments:

Listing 2.33:

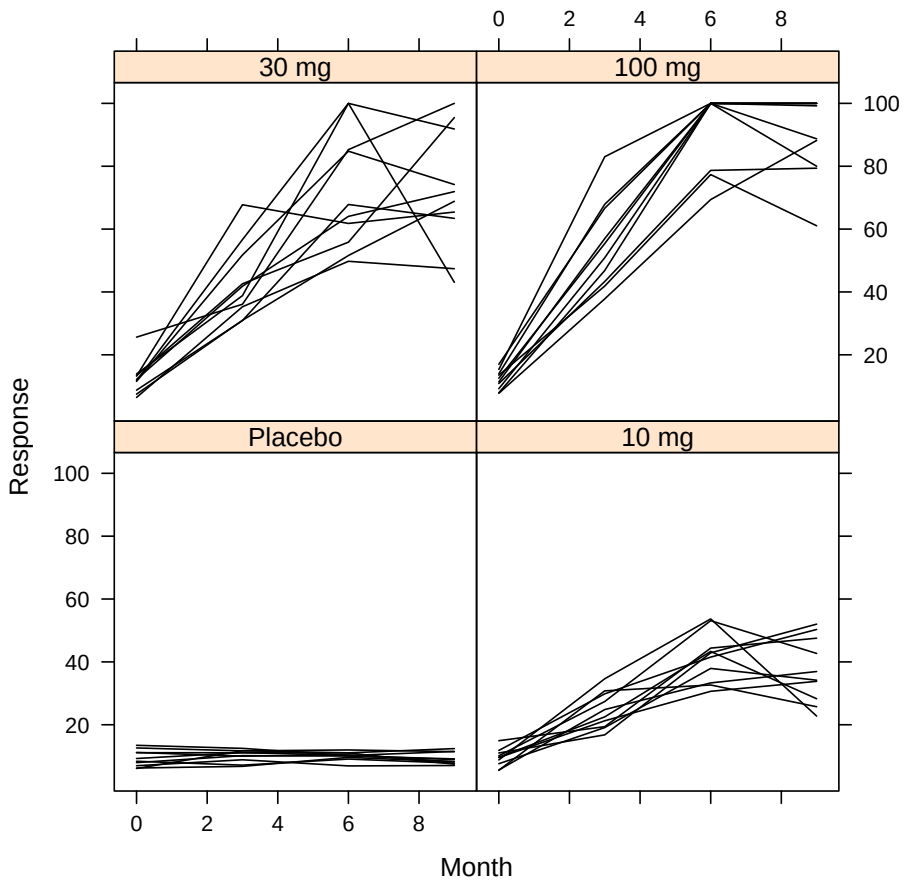
```
print(
      xyplot(Response ~ Month | Dose2,
             groups = Subject,
             data = dat,
             type = 'l'
             ## or type = 'b' if you want (b)oth points
             ## and lines
             )
      )
```



In this case, we are using the groups argument primarily to visually link multiple observations from the same subject; the ability to identify specific subjects is probably not so important. In that case, the color is more of a distraction than and aid to interpretation; we can use the same color for all groups by using the `col` argument.

Listing 2.34:

```
print(
      xyplot(Response ~ Month | Dose2,
             groups = Subject,
             data = dat,
             type = 'l',
             col = 'black'
             )
      )
```



In other cases, it's important to be able to specifically identify the groups from the plot. For example, suppose `Dose` is our grouping variable:

Listing 2.35:

```
print(
      xyplot(Response ~ Exposure | Month,
             groups = Dose,
             data = dat)
      )
```

We need a legend! There are extremely flexible ways of specifying the legend/key (flexible to the point were you can easily specify it incorrectly), but the easiest and safest way is to use the `auto.key` argument:

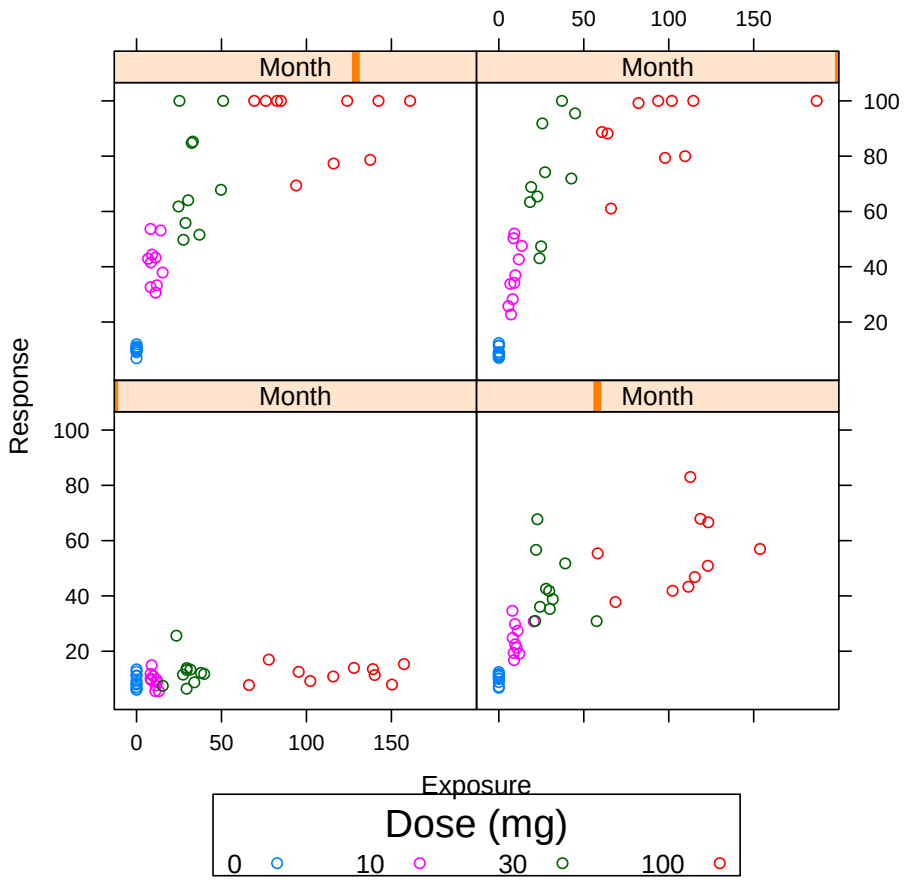
Listing 2.36:

```
print(
      xyplot(Response ~ Exposure | Month,
             groups = Dose,
             auto.key = TRUE,
             data = dat)
      )
```

You have some flexibility, even when working with `auto.key`:

Listing 2.37:

```
print(
      xyplot(Response ~ Exposure | Month,
             groups = Dose,
             auto.key = list(title = 'Dose (mg)', space = 'bottom
                ',
               columns = 4, border = TRUE),
             data = dat)
      )
```

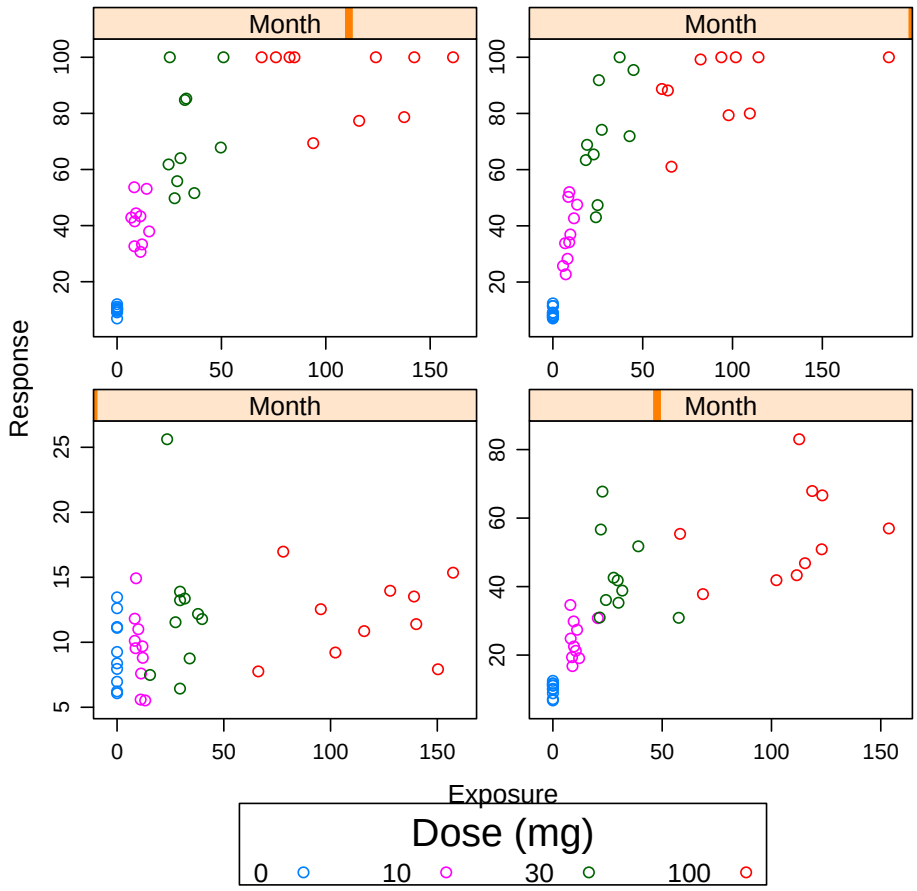


Using the scales argument

In all the trellised displays we have made so far, `lattice` has computed common axes and used them for every panel. That default is desirable when you want a "between-panel" perspective on the data. For example, our last plot makes it very clear that the placebo group had lower scores than any of the active dose groups for every post-baseline timepoint. On the other hand, sometimes you are more interested in "within-panel" variation. For example, might there be a trend in the placebo group over time? It's hard to tell from our last plot, because the scale in the placebo group panel might be dwarfing any trend that is there. One way of addressing this is by using the `scales` argument to establish a "free" relationship between the axes:

Listing 2.38:

```
print(
      xyplot(Response ~ Exposure | Month,
             groups = Dose,
             auto.key = list(title = 'Dose (mg)', space = 'bottom
                 ',
               columns = 4, border = TRUE),
             scales = list(relation = "free"),
             data = dat
             )
      )
```

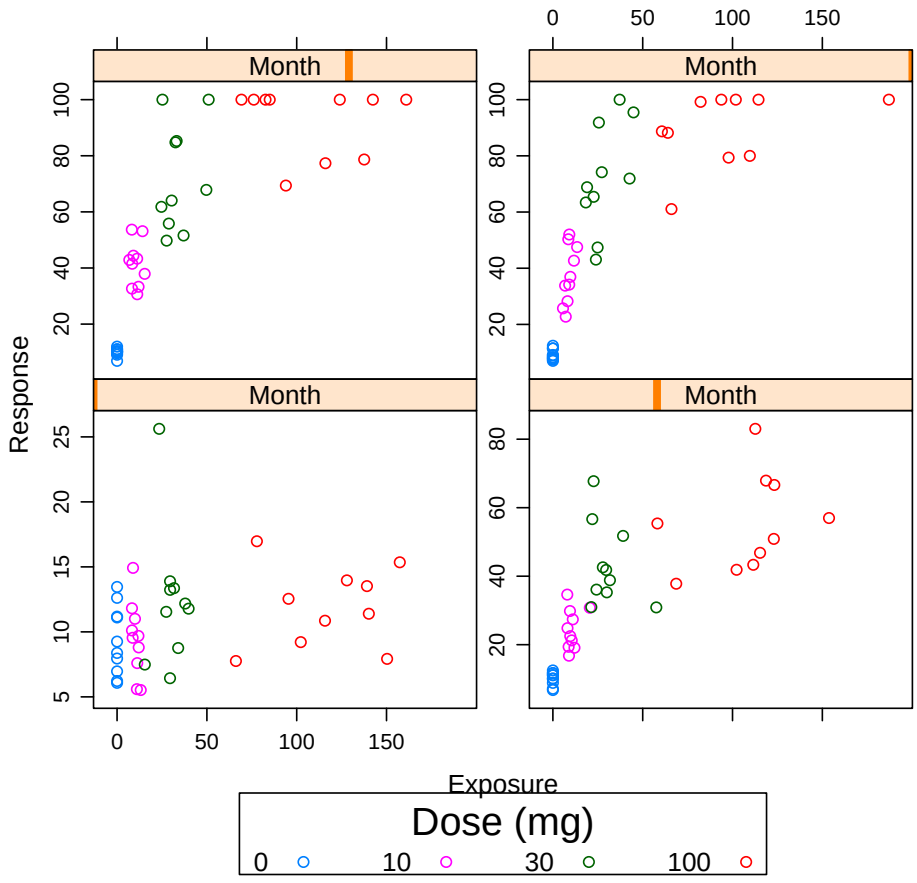


In this case there is no reason to compute the x axes separately; if we only want to free up the y axes, we can do:

Listing 2.39:

```
print(
```

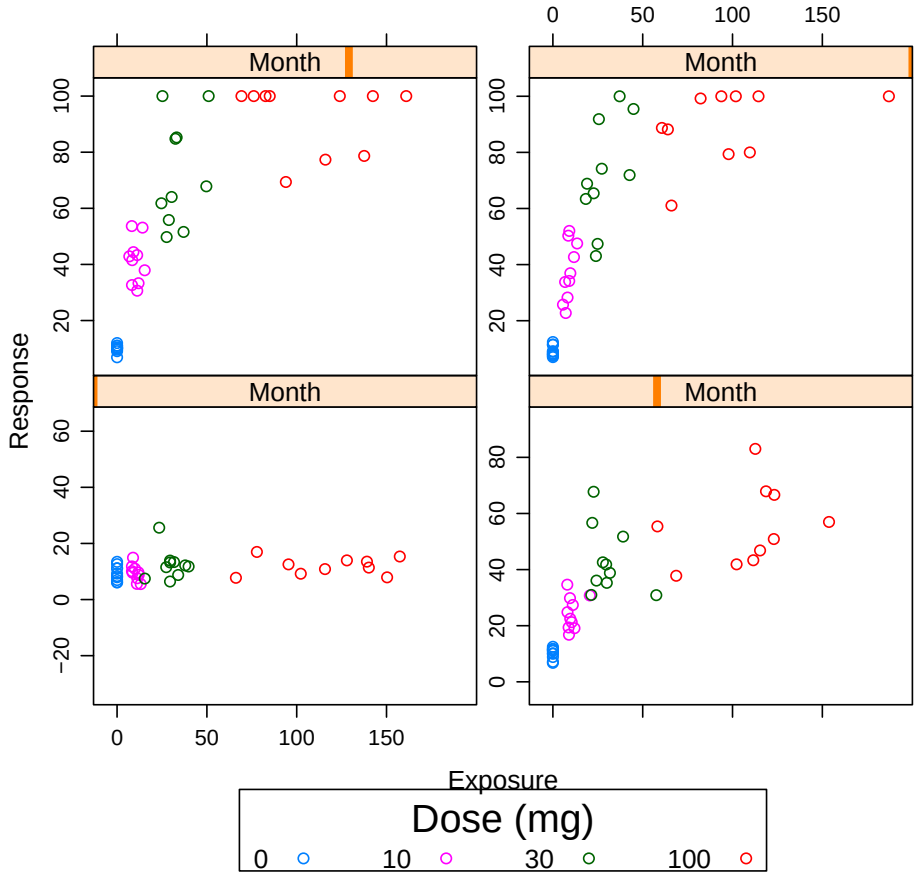
```
xyplot(Response ~ Exposure | Month,
       groups = Dose,
       auto.key = list(title = 'Dose (mg)', space = 'bottom
          ',
         columns = 4, border = TRUE),
       scales = list(
         x = list(relation = "same"),
         y = list(relation = "free")
         ),
       data = dat
       )
)
```



There is one other possible relation, "sliced", which makes all axes span ranges of equal length, but possibly centered at different values. It's not a particularly helpful perspective in this case, but we'll do it anyway for illustration.

Listing 2.40:

```
print(
      xyplot(Response ~ Exposure | Month,
             groups = Dose,
             auto.key = list(title = 'Dose (mg)', space = 'bottom
                 ',
               columns = 4, border = TRUE),
             scales = list(
               x = list(relation = "same"),
               y = list(relation = "sliced")
               ),
             data = dat
             )
      )
```



2.6 Homework

- Apply `xyplot` to this lesson's data set (`dat`) to show how the response varies with exposure within each dose group. Does exposure explain much of the variation in response, beyond what is already explained by dose?
- Make the same plot as previously, but using only data from month nine. Add a title to your plot that says "Month 9 Data Only".
- Save the plot you just made as a 3 inch by 3 inch figure in pdf format (we haven't discussed how to specify the plotting dimensions — you will need to search for help on the `pdf()` function)

Chapter 3

Basic Function Writing

3.1 Objectives

After completing this chapter, you will be able to . . .

- Distinquish between named and anonymous functions.
- Display the definition of a named function.
- Store and retrieve file-based function definitions.
- Write simple functions with one or more arguments.
- Write functions with default argument values.
- Write functions that accept arbitrary arguments.
- Explain the scope of function variables.

3.2 Introduction

Functions are the basic units of reusable code in R. If you find yourself writing the same instructions over and over, it may be time to capture your algorithm in a reusable function.

3.3 Simple Functions

Let's write a simple function that computes opposites.

Listing 3.1:

```
opposite <- function(x){
       return(-x)
}
```

This is a function declaration. Not all functions have names, but this one does: `opposite`. *Anonymous* functions are useful for passing to other functions; you'll use them later in the course.

The name is followed by the assignment operator and the reserved word `function`. The parentheses enclose the names of the function's formal *arguments*: information to be supplied by the user. Formal arguments are the declared arguments of a function, whereas *actual* arguments are those supplied when the function is called.

The curly braces enclose the function *body*: the list of instructions that comprise the algorithm. Every function body ends with a *return* value (exactly one!). In this case, we specify that the function returns the opposite of its sole argument. Let's try it.

Listing 3.2:

```
opposite(x=5)
opposite(x=5:10)
opposite(5:10)
opposite(opposite(5:10))
```

```
[1] -5
```

```
[1]  -5  -6  -7  -8  -9 -10
```

```
[1]  -5  -6  -7  -8  -9 -10
```

```
[1]  5  6  7  8  9 10
```

To use the function, we type its name, followed by arguments in parentheses. It is appropriate to name the arguments explicitly, but in this case it is not necessary: there is only one argument (`x`) and R can match what we pass to the formal argument name. Note that the return value of a function can itself be the argument to a function.

If you don't say `return` anywhere in the function body, the last expression evaluated is the returned value. Also, if the function body consists of only one expression, the curly braces are not needed. So a shorter definition of `opposite` is possible.

Listing 3.3:

```
opposite <- function(x)-x
```

In this case, the function body is `-x`. That is also the return value.

A function does not need to have any arguments (but the parentheses are still required). Here is an example.

Listing 3.4:

```
hello <- function()print('Hello World')
```

To view the definition of the function, simply enter its name at the prompt. To call this function, include the (empty) parentheses.

Listing 3.5:

```
hello
hello()
```

```
function ()
print("Hello World")

[1] "Hello World"
```

Sometimes you need to execute several steps before returning a value. Here, we write a function that removes all missing values from a vector, returning the (shortened) result.

Listing 3.6:

```
na.drop <- function(x){
       bad <- is.na(x)
       good <- x[!bad]
       return(good)
}
```

It's a good idea to save your function definitions in a text file, e.g, `myFunctions.R`. Then you can 'load' your functions simply.

Listing 3.7:

```
source('myFunctions.R')
```

3.3.1 Excercises

1. Create and save a file to store the functions you will write below.
2. Write a function that returns the first element of its argument.
3. Use `source()` to load your function definition(s).
4. Write a function that squares its argument.
5. Rewrite `na.drop` as a single expression (no curly braces).

6. Write a function that takes an integer and returns a day of the week.

7. Write a function that replaces all missing values in a vector with the mean of the non-missing values, returning the fully-defined vector.

3.4 Functions with Multiple Arguments

Many functions require two or more arguments. Addition, for example, requires left and right operands. In fact, the addition operator in R is really just another function. Here we view its definition, and even call it in an unconventional way.

Listing 3.8:

```
`+`
`+`(2,3)
```

```
function (e1, e2)  .Primitive("+")
```

```
[1] 5
```

Let's write a function that raises x to the power y.

Listing 3.9:

```
pow <- function(x,y)x**y
pow(2,3)
```

```
[1] 8
```

3.4.1 Exercises

1. Write a function that concatenates a protocol number and a subject ID, separated by a hyphen.

2. Write a function that normalizes weight by some typical value, rounding the result to 3 significant figures.

3. Write a function that normalizes `x` by its median, and rounds the result to `n` decimal places.

3.5 Default Argument Values

Normally it is an error to call a function without supplying all the necessary arguments. However, arguments often have default values, specified in the declaration. These need not be supplied. Let's revisit the previous example.

Listing 3.10:

```
pow <- function(x,y=2)x**y
pow(2,3)
pow(2)
pow(y=3)
```

```
[1] 8
```

```
[1] 4
```

```
'x' is missing
```

Default argument values can be expressions that depend on other arguments. For example, when calculating the area of a rectangle, we could assume by default that the rectangle is square.

Listing 3.11:

```
rectArea <- function(width,height=width)width*height
rectArea(width=3,height=4)
```

```
[1] 12
```

Listing 3.12:

```
rectArea(width=3)
```

```
[1] 9
```

3.6 Arbitrary Arguments

R allows functions that accept differing numbers of arguments. Normally, if an actual argument does not correspond to any formal argument, an error occurs. But if the special argument ... is present, it will be a list containing unmatched arguments, named or unnamed. Functions with ... arguments may operate directly on these arguments (eg. `paste` and `max`), may ignore them, or may pass them to other functions. The last behavior is often useful. For example, suppose you want a function that not only calculates the area of a rectangle, but also reports the units.

Listing 3.13:

```
specificRectArea <- function(..., units='meters')paste(
       rectArea(...),
       units
)
specificRectArea(3)
specificRectArea(3,units='feet')
```

```
[1] "9 meters"

[1] "9 feet"
```

3.6.1 Exercises

1. Write a function to evaluate the quadratic formula: $(-b \pm \sqrt{b^2 - 4ac})/2a$. Ignore unexpected arguments, and return the "additive" root by default.
2. Write a function to calculate AUC in terms of dose, clearance, and bioavailability. Assume bioavailability to be 1, if not given. (Clearance/bioavailability * AUC = dose.)
3. Suppose most of your data files are tab-separated, with headers, and have '.' to represent missing values. Write a 'wrapper' function for `read.table()` that sets these arguments, so that the caller only needs to supply a filename. Use ... to let the caller set other arguments. Is the caller able to override your value for 'header'?

3.7 Variable Scope

Scope is a technical term for the visibility of a defined variable. It is important to remember that variable changes and variable definitions made inside of functions are not visible in the "calling environment". In other words, "What happens here, stays here". Consider this example.

Listing 3.14:

```
do <- function(x){
       x <- 2 * x
       'done'
}
x <- 3
do(x)
x
```

```
[1] "done"
```

```
[1] 3
```

The variable x in the calling environment is distinct from the similarly-named variable x inside the function. Changes to x in the function have no lasting effect unless they are returned.

The reverse is not true, however. Variables defined in the calling environment may be visible in the function environment, even if not imported. So, "What happened there ... is visible here, too!"

Listing 3.15:

```
do <- function(){
       return(y * 2)
}
y <- 3
do()
```

```
[1] 6
```

Generally, it is dangerous to rely on this sort of visibility. Anything a function needs to know should be passed to it. There are cases, however, where this scoping behavior is useful.

3.8 Additional Exercises

1. Visit http://www.halls.md/ideal-weight/devine.htm and write an R function to calculate Miller's ideal body weight for men.
2. Same as above, but generalize the function to accommodate women as well.
3. Same as above, but assume subject is female if not specified.
4. What are all the possible ways you could calculate the square of 8, using pow() as defined in this chapter? Remember, actual arguments can be named or unnamed, in which case they are matched to the formal arguments by position.

Chapter 4

Data Summary

4.1 Objectives

After completing this chapter, you will be able to . . .

- Prepare R objects for summarizing
- Generate simple frequency tables using factors
- Understand the usage of the `apply()`, `lapply()`, `tapply()`, and `aggregate()` functions
- Subset dataframes using a `logical` vector

4.2 Introduction

After importing data, the next task that is usually performed is an exploration of the data. There are a number of simple, user-friendly functions in R for getting basic data summaries. These can occur at the individual variable level, at the intersection of two or more variables, across an entire data frame, etc. . . Some examples we have used in the Basic Mechanics section are shown below.

Listing 4.1:

```
library(metrumrg)
```

```
metrumrg 5.1
```

Listing 4.2:

```
prob1<-read.table(file="prob1.tab", skip=0, header=TRUE)
```

```
head(prob1)
```

```
  C ID    DV AMT II ADDL TIME RATE  HT WT   CLCR SEX AGE
1 0  1  0.00 900 12    9  0.0  300 103 91 101.48   0  46
2 0  1  6.99   0  0    0  1.5    0 103 91 101.48   0  46
3 0  1 12.66   0  0    0  3.0    0 103 91 101.48   0  46
4 0  1  5.31   0  0    0  8.0    0 103 91 101.48   0  46
5 0  1  2.39   0  0    0 11.9    0 103 91 101.48   0  46
6 0  1  2.87   0  0    0 23.9    0 103 91 101.48   0  46
```

Listing 4.3:

```
median(prob1$DV)
```

```
[1] 3.52
```

Listing 4.4:

```
mean(prob1$DV)
```

```
[1] 4.2787
```

Listing 4.5:

```
summary(prob1[prob1$SEX==0, c("DV")])
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  0.000   2.120   3.400   4.016   5.310  16.560
```

Listing 4.6:

```
summary(prob1)
```

```
       C             ID              DV               AMT
 Min.   :0   Min.   : 1.00   Min.   : 0.000   Min.   :  0
 1st Qu.:0   1st Qu.:10.75   1st Qu.: 2.127   1st Qu.:  0
 Median :0   Median :20.50   Median : 3.520   Median :  0
 Mean   :0   Mean   :20.50   Mean   : 4.279   Mean   : 60
 3rd Qu.:0   3rd Qu.:30.25   3rd Qu.: 5.930   3rd Qu.:  0
 Max.   :0   Max.   :40.00   Max.   :21.280   Max.   :900
       II             ADDL           TIME
 Min.   : 0.0   Min.   :0.0   Min.   :  0.00
 1st Qu.: 0.0   1st Qu.:0.0   1st Qu.:  8.00
 Median : 0.0   Median :0.0   Median : 47.90
 Mean   : 0.8   Mean   :0.6   Mean   : 60.82
 3rd Qu.: 0.0   3rd Qu.:0.0   3rd Qu.:111.00
 Max.   :12.0   Max.   :9.0   Max.   :144.00
      RATE            HT               WT
```

```
Min.   :  0   Min.   : 41.00   Min.   :34.00
1st Qu.:  0   1st Qu.: 59.25   1st Qu.:50.50
Median :  0   Median : 69.00   Median :58.00
Mean   : 20   Mean   : 70.33   Mean   :60.52
3rd Qu.:  0   3rd Qu.: 79.00   3rd Qu.:70.25
Max.   :300   Max.   :108.00   Max.   :96.00
     CLCR             SEX             AGE
Min.   : 24.58   Min.   :0.000   Min.   :38.00
1st Qu.: 47.70   1st Qu.:0.000   1st Qu.:44.00
Median : 58.94   Median :0.000   Median :45.50
Mean   : 66.54   Mean   :0.425   Mean   :45.65
3rd Qu.: 72.89   3rd Qu.:1.000   3rd Qu.:47.25
Max.   :192.38   Max.   :1.000   Max.   :51.00
```

While the commands above are nice for getting quick results, they can be cumbersome and error-prone to apply in many everyday situations. This would include summarizing a variable by a grouping variable that has more than two categories, creating summaries at the level of the individual, handling missing data, and utilizing summary results as an input to one or more additional functions. We can tweak the `prob1` data set to make it more interesting for summarizing by adding a few additional columns.

Listing 4.7:

```
set.seed(35241)
prob1<-read.table(file="prob1.tab", skip=0, header=TRUE)
IDs <- unique(prob1$ID)
race <- sample(c("Caucasian","Asian","African American","Other"),
   40, replace=T)
food <- sample(c(0:2), 40, replace=T)
wt2 <- rnorm(40, mean=5, sd=2)
racefoodwt2 <- data.frame("ID" = IDs, "RACE" = race, "FOOD" = food
   , "WT2" = wt2)
prob1 <- merge(prob1, racefoodwt2, by="ID")
prob1$WT2 <- prob1$WT+prob1$WT2
prob1$WT3 <- prob1$WT2
prob1$WT3[prob1$WT3>80] <- NA
prob1$RACE <- as.factor(prob1$RACE)
prob1$SEX <- as.factor(prob1$SEX)
prob1$FOOD <- as.factor(prob1$FOOD)
summary(prob1)
```

```
      ID              C           DV              AMT
Min.   : 1.00   Min.   :0   Min.   : 0.000   Min.   :  0
1st Qu.:10.75   1st Qu.:0   1st Qu.: 2.127   1st Qu.:  0
Median :20.50   Median :0   Median : 3.520   Median :  0
Mean   :20.50   Mean   :0   Mean   : 4.279   Mean   : 60
```

```
3rd Qu.:30.25   3rd Qu.:0   3rd Qu.: 5.930   3rd Qu.:  0
Max.   :40.00   Max.   :0   Max.   :21.280   Max.   :900

      II            ADDL          TIME
Min.   : 0.0   Min.   :0.0   Min.   :  0.00
1st Qu.: 0.0   1st Qu.:0.0   1st Qu.:  8.00
Median : 0.0   Median :0.0   Median : 47.90
Mean   : 0.8   Mean   :0.6   Mean   : 60.82
3rd Qu.: 0.0   3rd Qu.:0.0   3rd Qu.:111.00
Max.   :12.0   Max.   :9.0   Max.   :144.00

     RATE           HT              WT
Min.   :  0   Min.   : 41.00   Min.   :34.00
1st Qu.:  0   1st Qu.: 59.25   1st Qu.:50.50
Median :  0   Median : 69.00   Median :58.00
Mean   : 20   Mean   : 70.33   Mean   :60.52
3rd Qu.:  0   3rd Qu.: 79.00   3rd Qu.:70.25
Max.   :300   Max.   :108.00   Max.   :96.00

     CLCR         SEX          AGE
Min.   : 24.58   0:345   Min.   :38.00
1st Qu.: 47.70   1:255   1st Qu.:44.00
Median : 58.94           Median :45.50
Mean   : 66.54           Mean   :45.65
3rd Qu.: 72.89           3rd Qu.:47.25
Max.   :192.38           Max.   :51.00

               RACE     FOOD         WT2
African American:210   0:180   Min.   :37.87
Asian           :135   1:165   1st Qu.:56.10
Caucasian       :150   2:255   Median :63.44
Other           :105           Mean   :65.87
                               3rd Qu.:74.90
                               Max.   :99.01

     WT3
Min.   :37.87
1st Qu.:54.26
Median :61.88
Mean   :61.41
3rd Qu.:70.56
Max.   :78.52
NA's   :90.00
```

From the `summary(prob1)` step we now see this data set contains three factor level variables (RACE, SEX, and FOOD) and two additional weight variable (WT2 and WT3). We will use this revised `prob1` data set for the remainder of this chapter.

4.3 Simple Summaries Using table()

We can begin summarizing interesting portions of the `prob1` data set using various R functions. The first function we will use is `table()`. This function will generate a frequency table of the counts at each combination of factor levels. This will enable us to get a picture of male/female counts, male/female counts across RACE categories, male/female counts across FOOD categories, FOOD counts across RACE categories, etc...

Listing 4.8:

```
table(prob1$SEX)
```

```
  0   1
345 255
```

Listing 4.9:

```
table(prob1$SEX, prob1$RACE)
```

```
    African American Asian Caucasian Other
  0              105    90        75    75
  1              105    45        75    30
```

A more informative approach would be to limit the data set to one line per individual prior to creating the summary. We can use the `[` operator and `duplicated()` function to subset our data 'on the fly'.

Listing 4.10:

```
table(prob1[!duplicated(prob1$ID), c("SEX")])
```

```
 0  1
23 17
```

Listing 4.11:

```
table(prob1[!duplicated(prob1$ID), c("SEX","RACE")])
```

```
   RACE
SEX African American Asian Caucasian Other
  0                7     6         5     5
  1                7     3         5     2
```

Listing 4.12:

```
table(prob1[!duplicated(prob1$ID), c("SEX","FOOD")])
```

```
   FOOD
SEX  0  1  2
  0  5  7 11
  1  7  4  6
```

Listing 4.13:

```
table(prob1[!duplicated(prob1$ID), c("FOOD","SEX","RACE")])
```

```
, , RACE = African American

    SEX
FOOD 0 1
   0 1 2
   1 5 3
   2 1 2

, , RACE = Asian

    SEX
FOOD 0 1
   0 2 1
   1 1 0
   2 3 2

, , RACE = Caucasian

    SEX
FOOD 0 1
   0 1 4
   1 0 0
   2 4 1

, , RACE = Other

    SEX
FOOD 0 1
   0 1 0
   1 1 1
   2 3 1
```

The summaries utilizing non-duplicated `ID` values results in more meaningful counts. Given the results of the category counts, we could begin to formalize decisions around what type

of questions could be answered or not with the current data set. For example, we have a reasonable split between males and females so we may be able to ask/answer, *What are the effects of gender on clearance?*. We may also be able to ask/answer the question, *What are the effects of FOOD on clearance?*. However, due to the small sample size (n=40), we may not want to try and answer the question, *What are the effects of FOOD given gender on clearance?*. Whether these questions are scientifically/clinically relevant also needs to be considered but is not something we will address here.

4.3.1 Exercises

- load the `ChickWeight` data set and determine the number of weight observations per Diet.
- How many weight observations per Chick?

4.4 Applying Summary Functions Across Vectors and Dataframes

The use of `table()` has reduced the amount of R code to get factor level summaries but does not provide a result that is easily used as input to other functions. R provides a very powerful set of inter-related functions, `apply()`, `tapply()`, `lapply()`, and `aggregate()`, that can be used to "apply" a function to an R object. Lets start by looking at the arguments that are required by `apply()` utilizing the `args()` function.

Listing 4.14:

```
args(apply)
```

```
function (X, MARGIN, FUN, ...)
NULL
```

We will need to supply "X", "MARGIN", and "FUN" arguments to `apply`. If we were unsure of the data requirements for these arguments we could use R-help for additional information. We would find out that "X" should represent the data to be used, "MARGIN" indicates how to apply the function, e.g., `1` indicates rows and `2` indicates columns, and "FUN" is the function to apply. We will utilize available R functions for the "FUN" argument but a user written function could also be used.

Listing 4.15:

```
apply(X=prob1[!duplicated(prob1$ID),c("WT","CLCR")], MARGIN=2, FUN
   =mean)
```

```
      WT     CLCR
60.52500 66.54375
```

Listing 4.16:

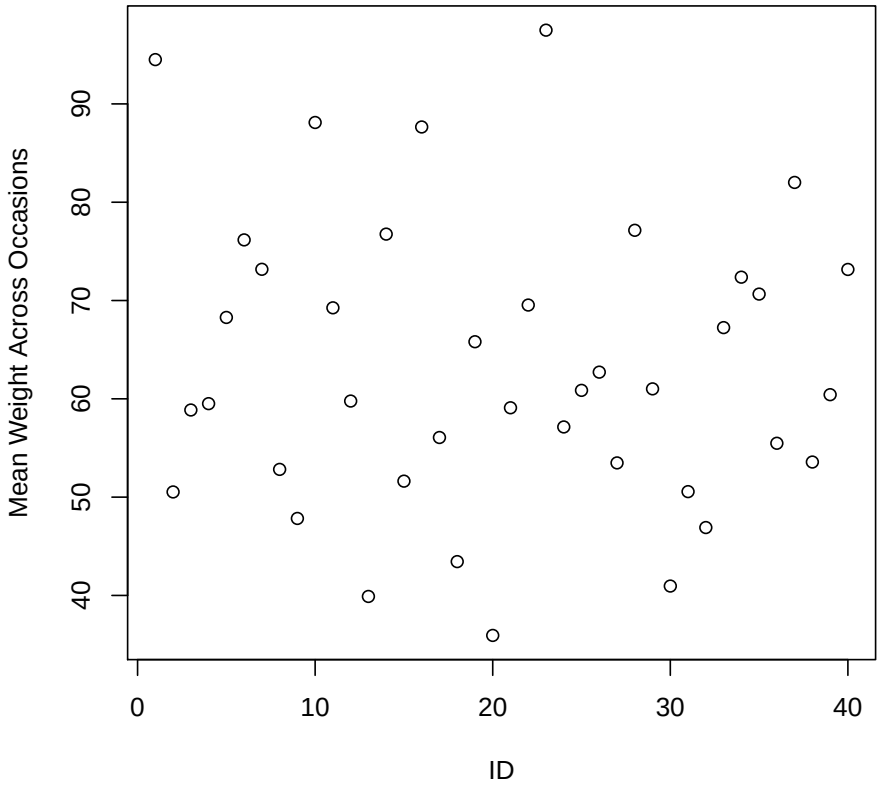
```
apply(X=prob1[!duplicated(prob1$ID),c("WT","WT2")], MARGIN=1, FUN=
   mean)
```

```
       1        16        31        46        61        76
94.50746 50.52943 58.86600 59.51227 68.27993 76.16969
      91       106       121       136       151       166
73.17902 52.82397 47.83453 88.11308 69.25926 59.77521
     181       196       211       226       241       256
39.90034 76.75797 51.62801 87.65498 56.06839 43.44307
     271       286       301       316       331       346
65.80988 35.93439 59.09193 69.54200 97.50347 57.13954
     361       376       391       406       421       436
60.86345 62.71663 53.48403 77.14603 61.01500 40.95554
     451       466       481       496       511       526
50.56214 46.90958 67.24608 72.37702 70.65591 55.48285
     541       556       571       586
82.01322 53.57233 60.42317 73.16164
```

The first use of `apply` returns the mean weight and creatinine clearance across the `prob1` data set while the second example returns the mean weight of each individual. We achieve the latter result by applying the function across the rows (`MARGIN 1`) rather than per column (`MARGIN 2`). The output of both `apply` arguments is a simple vector with `names` corresponding to the column or row names used in the calculation. We could have used the output for preparing a plot as the example below shows.

Listing 4.17:

```
wt.ind <- apply(X=prob1[!duplicated(prob1$ID),c("WT","WT2")],
   MARGIN=1, FUN=mean)
plot(x=unique(prob1$ID),
     y=wt.ind,
     xlab="ID",
     ylab = "Mean Weight Across Occasions")
```



Now, lets take a look at the variants of `apply`. The `lapply` function takes only the `X` and `FUN` arguments. It applies the `FUN` argument to each element in `X` and returns a list of the same length as `X`. This function simplifies the R code but is limited to operating on columns of `X`. How could we use `lapply` to calculate the maximum weight (WT or WT3) and DV for `prob1`?

Listing 4.18:

```
prob1.ind <- prob1[!duplicated(prob1$ID),]
lapply(X=prob1.ind[ ,c("DV","WT")], FUN = max)
```

```
$DV
[1] 0

$WT
[1] 96
```

Listing 4.19:

```
lapply(X=prob1[ ,c("DV","WT")], FUN = max)
```

```
$DV
[1] 21.28

$WT
[1] 96
```

Listing 4.20:

```
lapply(X=prob1[ ,c("DV","WT3")], FUN = max)
```

```
$DV
[1] 21.28

$WT3
[1] NA
```

The first result of `lapply` may not be what we expected. We were attempting to determine the maximum weight and DV value in `prob1`. In the first evaluation, we used an R object that was limited to the first row in each individual resulting in a 0 value for DV for all individuals. We wanted to evaluate the `max` function across the entire data set as we did in the second example. The third example demonstrates why it is important to understand the data prior to summarizing. The `WT3` variable contains some missing (NA) values so the result of applying `max` over this variable is "NA". We could have passed `na.rm=TRUE` as an additional argument to `FUN` so "NA" values would be removed prior to applying the `max` function. NOTE: All of the `FUN` arguments will accept additional arguments that are specific to the function being used.

The last variant of `apply` we will look at is `tapply`. This function takes the `X`, `INDEX`, `FUN`, and `simplify` arguments. `tapply` expects `X` to be a vector and will error if any other type of object is passed to it. `tapply` applies `FUN` to `X` by the unique combination of levels of `INDEX`. We can think of `tapply` as tying together the `FUN` argument and the `table` results. Lets take a look at how we could use `tapply` to calculate the maximum DV value for each subject.

Listing 4.21:

```
maxDV <- tapply(X=prob1$DV, INDEX=prob1[,c("ID")], FUN=max)
head(maxDV)
```

```
    1     2     3     4     5     6
12.66 15.68 13.46 11.61  8.35 13.03
```

Listing 4.22:

```
maxDV <- tapply(X=prob1$DV, INDEX=prob1[,c("ID")], FUN=max,
   simplify=FALSE)
head(maxDV)
```

```
$`1`
[1] 12.66

$`2`
[1] 15.68

$`3`
[1] 13.46

$`4`
[1] 11.61

$`5`
[1] 8.35

$`6`
[1] 13.03
```

Both examples calculate the maximum DV value for each individual and then list the first six results in the `maxDV` object. The second call to `tapply` utilizes the `simplify` argument to produce a `list` object rather than a `vector` object. The choice of TRUE or FALSE for the `simplify` argument will be guided by how the resulting R object will be used.

R also provides a function called `aggregate` that combines calls to `tapply` and `lapply` across a `data.frame`. `aggregate` splits the data into subsets, computes summary statistics for each, and returns the results in a convenient form. We can use the `args` function to determine the arguments and then use this information to compute the maximum DV by ID and VISIT.

Listing 4.23:

```
args(aggregate.data.frame)
```

```
function (x, by, FUN, ...)
NULL
```

Listing 4.24:

```
prob2<-prob1
prob1$VISIT<-1
prob2$VISIT<-5
prob2$DV <- prob2$DV*5
prob1 <- rbind(prob1, prob2)
maxDVbyVISIT <- aggregate.data.frame(x=prob1[,c("DV")], by=list(
   prob1$VISIT, prob1$ID), FUN=max)
head(maxDVbyVISIT, n=12)
```

```
   Group.1 Group.2     x
1        1       1 12.66
2        5       1 63.30
3        1       2 15.68
4        5       2 78.40
5        1       3 13.46
6        5       3 67.30
7        1       4 11.61
8        5       4 58.05
9        1       5  8.35
10       5       5 41.75
11       1       6 13.03
12       5       6 65.15
```

The results above demonstrate the maximum DV value by VISIT for each individual. `aggregate` returns a `data.frame` with column names of "Group.1" to "Group.x", where the ordering and number of columns corresponds to the variables in the `by` argument and an "x" column that contains the results of the applied `FUN`. Additional arguments specific to the `FUN` function can be passed in the same manner as we did above for the `apply` functions.

4.4.1 Exercises

- load the `sleep` data set and determine the average sleep increase in hours by drug.
- load the `women` data set and determine the average height and weight using `apply` and `lapply`.

4.5 Subsetting a dataframe using a logical vector

A `logical` vector in R contains TRUE or FALSE values for all of its elements. It can be created with the `logical` function but most of the time it is generated by "testing" a variable. For example, the code below tests whether the `WT3` variable contains any NA values then prints out all of the "unique" values in the `testNA` R object.

Listing 4.25:

```
testNA <- is.na(prob1$WT3)
unique(testNA)
```

```
[1]  TRUE FALSE
```

Listing 4.26:

```
length(testNA)
```

```
[1] 1200
```

The `length` of `testNA` is equal to the number of rows in `prob1`. (How could we verify the number of rows in `prob1` ?) We now have a `logical` vector that can be used to subset our data set based on the presence or absence of NA values in `WT3`.

Listing 4.27:

```
prob3 <- prob1[testNA,]
dim(prob3)
```

```
[1] 180  18
```

Listing 4.28:

```
unique(prob3$WT3)
```

```
[1] NA
```

Listing 4.29:

```
prob4 <- prob1[!testNA,]
dim(prob4)
```

```
[1] 1020   18
```

Listing 4.30:

```
unique(prob4$WT3)
```

```
 [1] 54.05885 61.73200 62.02454 70.55987 78.33937 75.35804
 [7] 56.64795 50.66906 73.51853 62.55041 40.80069 78.51593
[13] 54.25601 58.13678 45.88614 69.61976 37.86879 59.18386
[19] 72.08399 59.27908 64.72690 65.43326 55.96807 63.03001
[25] 41.91107 52.12429 50.81915 70.49216 74.75403 74.31182
[31] 57.96570 56.14467 63.84633 75.32328
```

We can see from the results above that the use of a `logical` vector is a very simple and concise way to generate a subset.

4.6 Homework

- Read in the `data5.csv` data set and create a new dataframe containing the average DV and CRCL by DAY.

- Using the same `data5.csv` data set calculate the median and maximum WT by DAY. Be careful, there are missing values (`NA`'s) in the WT variable. How can we deal with these?
- In many of the examples in this chapter, we used the `duplicated` function with the logical `!` operator to limit `prob1` to one row per individual prior to performing data summaries. Generate a `logical` vector that could be used to subset `prob1` to one line per individual and determine minimum and maximum CRCL in this subset of `prob1`.
- Load the `women` data set. Write a function that converts weight in pounds to weight in kilograms. Convert the `weight` variable in the `women` data set from pounds to kilograms using your function with `lapply`.

Chapter 5

Advanced Plotting

5.1 Objectives

After completing this chapter, you will be able to . . .

- Understand the role of the `panel` argument to `xyplot()`.
- Write your own `panel` functions.
- Understand how the function `panel.superpose()`, the argument `panel.groups`, and the lattice `theme` interact.

5.2 Introduction

We have already seen how `xyplot()` is a very *convenient* function to use, e.g. multi-panel plots can be created by just introducing a conditioning variable in your formula. In this chapter we are going to see that `xyplot()` is also very *extensible*. The cornerstone of this extensibility is the `panel` argument.

Let's recreate the same data set we used in Chapter 2 (but this time adding in a gender variable).

Listing 5.1:

```
set.seed(1)
nSubj <- 40
doses <- c(0, 10, 30, 100)
times <- seq(0, 9, 3)
dat <- expand.grid(Subject = 1:nSubj,
                   Month = times)
```

```
dat <- as.data.frame(unclass(dat))
dat$Dose <- doses[as.numeric(dat$Subject) %% length(doses) + 1]
dat$Exposure <- dat$Dose * exp(rnorm(nrow(dat), 0, 0.3))
noise <- rep(rnorm(nSubj, 0, 0.2), length(doses)) + rnorm(nrow(dat
   ), 0, 0.2)
emax <- c(0, 50, 100, 100)[match(dat$Month, times)]
dat$Response <- with(dat, 10 + emax * Exposure / (Exposure + 25))
   * exp(noise)
dat$Response <- pmin(dat$Response, 100)
dat$Dose2 <- factor(paste(dat$Dose, "mg"))
dat$Dose2 <- factor(dat$Dose2, levels = levels(dat$Dose2)[c(1, 2,
   4, 3)])
dat$Sex <- rep(sample(c("M", "F"), nSubj, replace = TRUE), length(
   times))
```

And before we get started let's not forget to load the lattice package:

Listing 5.2:

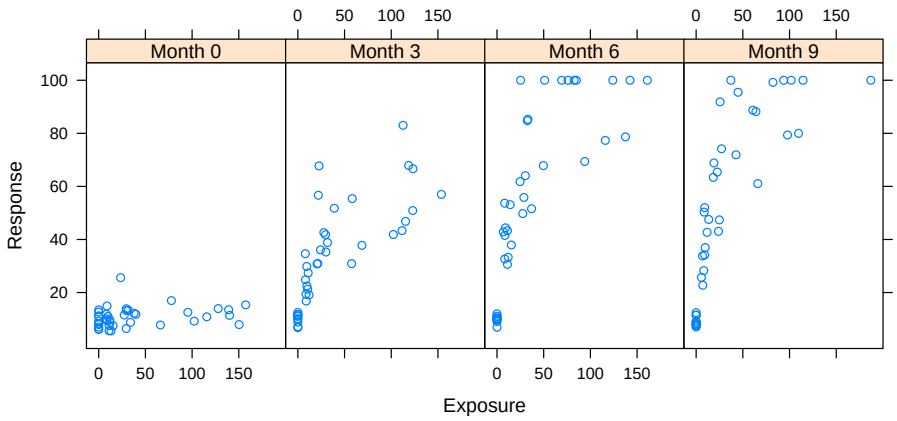
```
library(lattice)
```

5.3 The panel argument

The `panel` argument to `xyplot()` is used to tell `xyplot()` what to do inside each panel. In our examples in Chapter 2, we never explicitly specified a value for `panel`, so it just defaulted to `panel.xyplot`. We could have made this default explicit by doing:

Listing 5.3:

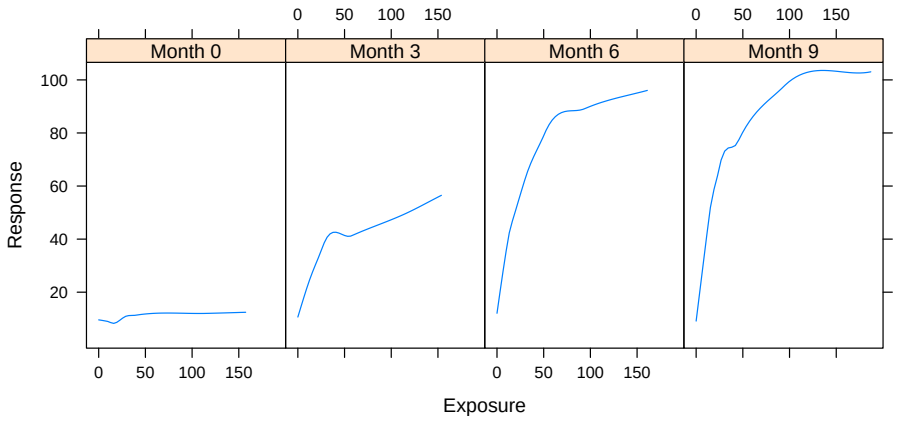
```
print(
xyplot(Response ~ Exposure | paste("Month", Month),
       data = dat,
       layout = c(4, 1),
       panel = panel.xyplot,
       )
      )
```



By specifying `panel = panel.xyplot`, we are basically saying, "just make a basic scatterplot inside each panel". If we want something more exciting to happen within the panels, we will need to change the value supplied for this argument. There are some built-in panel functions (see `?panel.functions`) that we could use instead of `panel.xyplot`, but most of them are not that helpful on their own. For, example, `panel.loess` draws a "loess" fit to the data:

Listing 5.4:

```
print(
xyplot(Response ~ Exposure | paste("Month", Month),
       data = dat,
       layout = c(4, 1),
       panel = panel.loess,
       )
      )
```



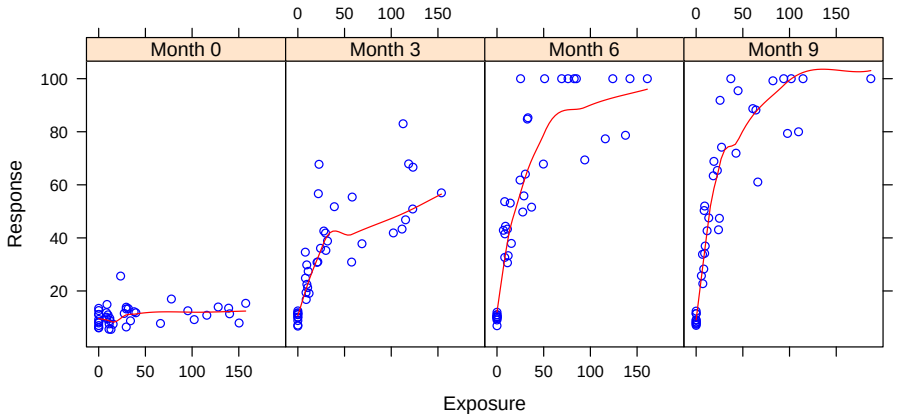
But what happened to our data points? If we want both the data points and the loess fit to be shown, we have to write our own panel function. Let's do it first and think about it later:

Listing 5.5:

```
my.panel <- function(x, y,
                     my.point.color = "blue",
                     my.loess.color = "red") {
  panel.xyplot(x, y, col = my.point.color)
  panel.loess(x, y, col = my.loess.color)
  return(NULL)
}
```

Listing 5.6:

```
print(
xyplot(Response ~ Exposure | paste("Month", Month),
       data = dat,
       layout = c(4, 1),
       panel = my.panel,
       )
      )
```



Let's notice a few things about our custom panel function:

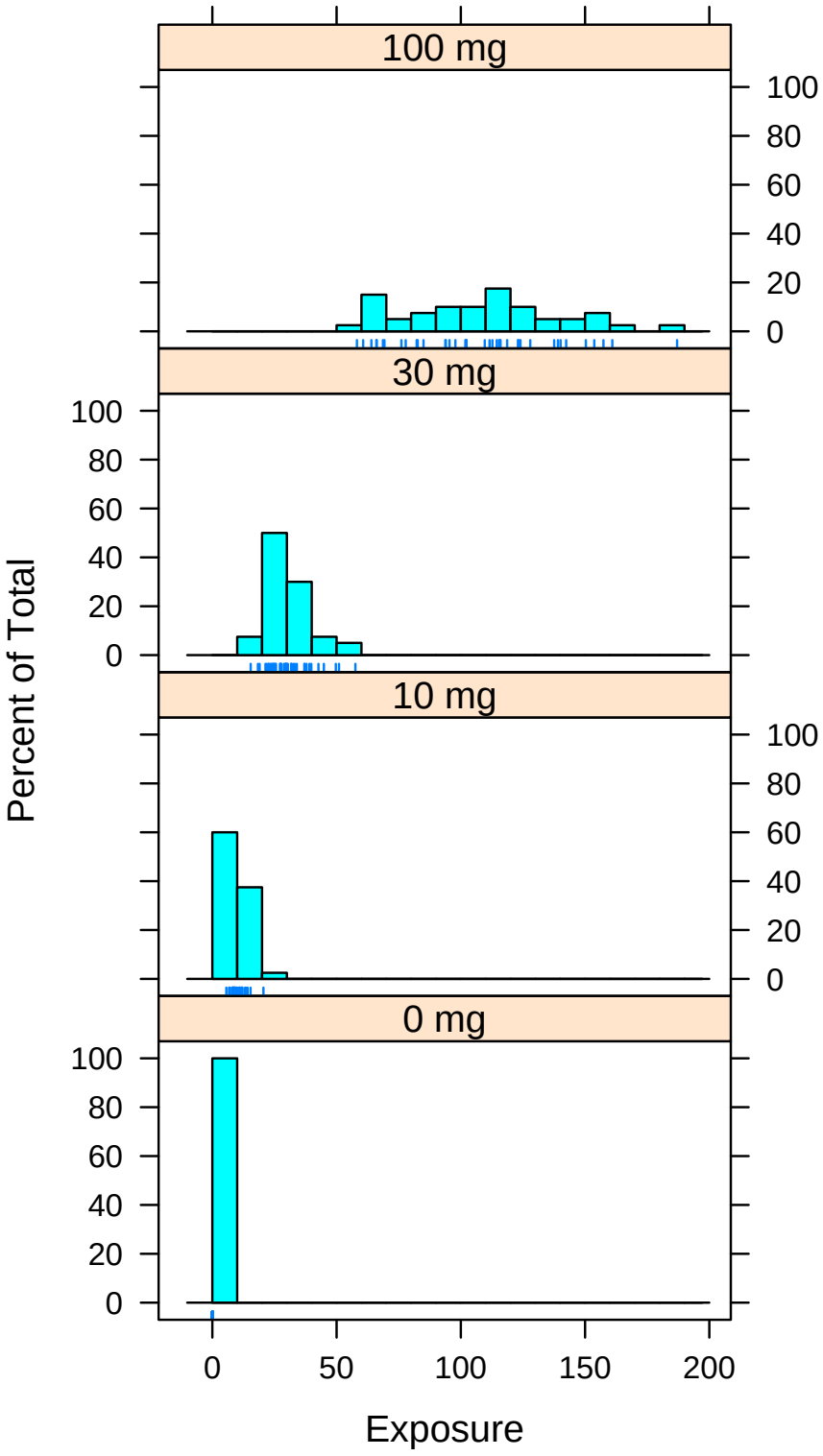
- We never called `my.panel()` directly. We called `xyplot()`, and while it was executing it called `my.panel()`.
- We can see that `xyplot()` must have called `my.panel()` four different times, once for each panel.
- We used the names `x` and `y` for the first two arguments of `my.panel()`. This naming convention is important because `xyplot()` has to know how to set up its call to `my.panel()`. Behind the scenes `xyplot()` does things like: `my.panel(x = Exposure[Month`

- The return value of our panel function (`NULL`) is useless. The drawing of graphics that takes place is considered a "side effect" of the function, not a return value. Panel functions are invoked for their side effects, not to compute return values. (There is no need to explicitly specify a return value at all, although one could argue that always providing an explicit return value for all functions that you write makes your code more readable).

We have been focusing on `xyplot()`, but note that most other high level lattice functions also take a panel argument. For example, the default panel function for `histogram()` is `panel.histogram()`, but as with `xyplot()`, we can easily modify this default. If you prefer you can just define your panel function on the fly, as in this example:

Listing 5.7:

```
print(
histogram( ~ Exposure | Dose2,
          data = dat,
          layout = c(1, 4),
          panel = function(x) {
            panel.histogram(x, breaks = seq(0, 200, 10), type = '
               percent')
            panel.rug(x)
          }
         )
      )
```

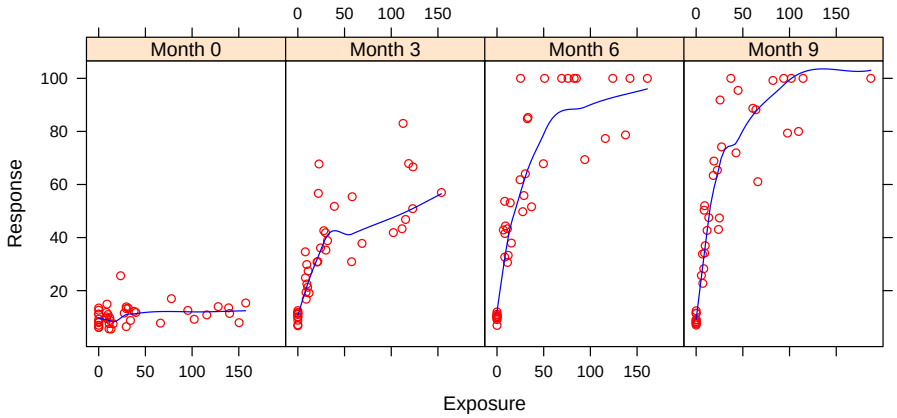

100 mg
30 mg
10 mg
0 mg
Percent of Total
Exposure
0
20
40
60
80
100
0
50
100
150
200

5.4 Passing information to your panel function

We have already seen how values are supplied to the canonical arguments `x` and `y` of our panel function. What about the other arguments? What if we want to override the default values that we specified when we wrote the function? One thing we can do is supply them as arguments to the high-level function:

Listing 5.8:

```
print(
xyplot(Response ~ Exposure | paste("Month", Month),
       data = dat,
       layout = c(4, 1),
       panel = my.panel,
       my.point.color = "red",
       my.loess.color = "blue"
       )
      )
```



It is also very often convenient to pass information along using the special `...` argument. In the following example, the argument `lty` is passed to `my.panel` and subsequently to `panel.loess` as a `...` argument.

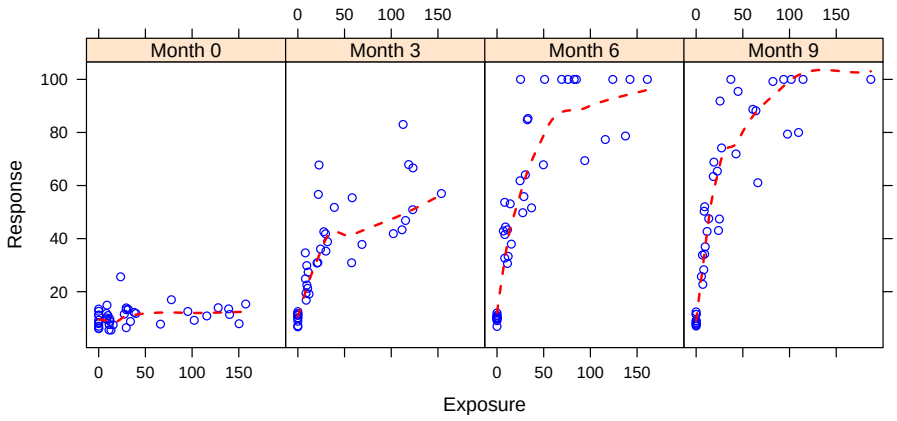
Listing 5.9:

```
my.panel <- function(x, y,
                     my.point.color = "blue",
                     my.loess.color = "red",
                     ...
                     ## "..." is like saying "and
                     ## whatever else I didn't think of"
                     ) {
```

```
  panel.xyplot(x, y, col = my.point.color)
  panel.loess(x, y, col = my.loess.color, ...)
  return(NULL)
}
```

Listing 5.10:

```
print(
xyplot(Response ~ Exposure | paste("Month", Month),
       data = dat,
       layout = c(4, 1),
       panel = my.panel,
       my.point.color = "blue",
       my.loess.color = "red",
       lty = 2, # gets passed to my.panel as "...",
       lwd = 2
       )
      )
```



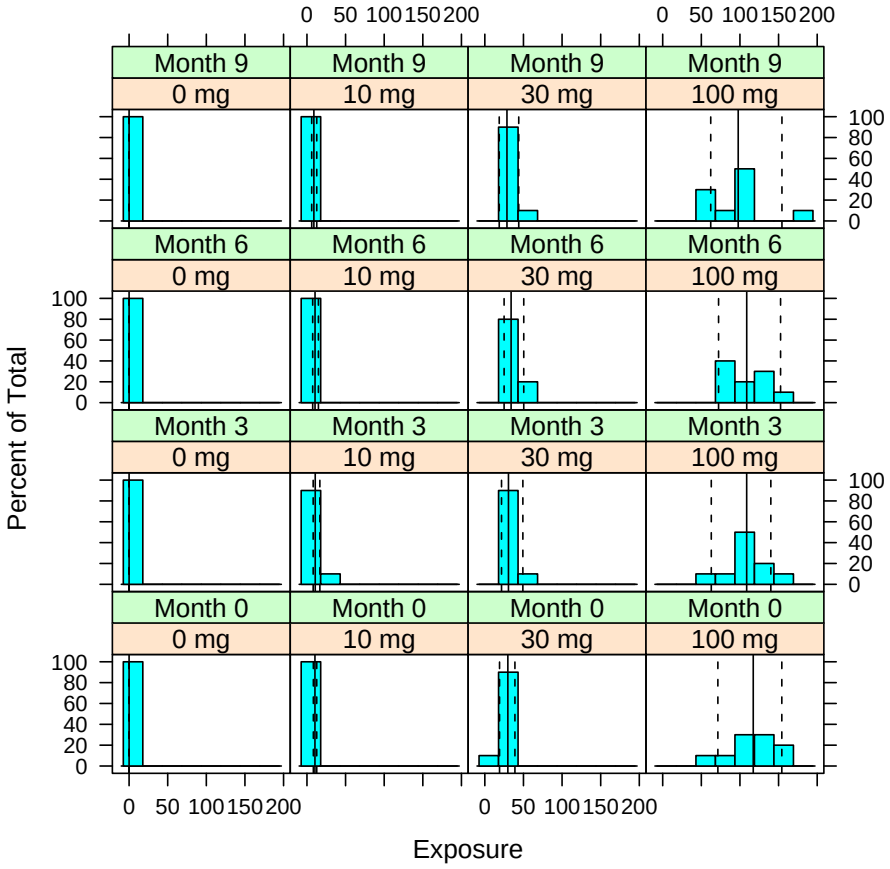
5.5 Doing computations inside your panel function

While the ultimate purpose of a panel function is to draw stuff, there is no reason we can't do computations within our panel function as well. While we're at it, let's see what happens when we condition on two variables:

Listing 5.11:

```
print(
histogram( ~ Exposure | Dose2 * paste("Month", Month),
          data = dat,
```

```
        panel = function(x, ...) {
          panel.histogram(x, ...)
          xmean <- mean(x)
          xquantile <- quantile(x, probs = c(0.05, 0.95))
          panel.abline(v = xmean)
          panel.abline(v = xquantile, lty = 2)
        }
      )
    )
```



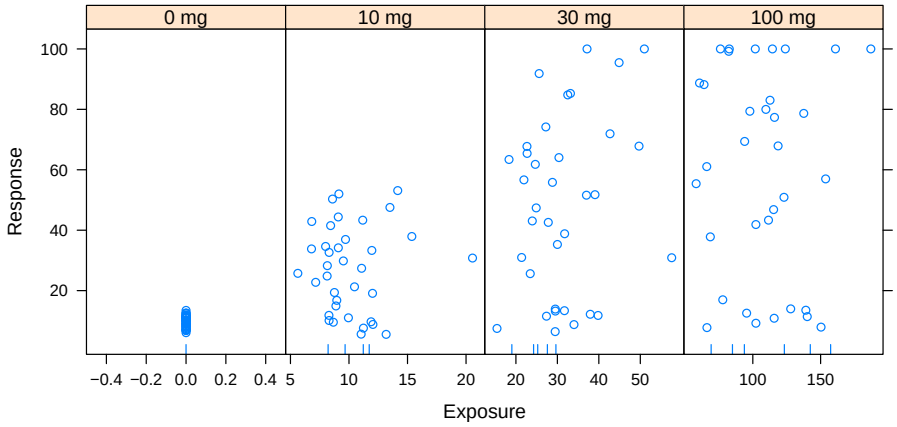
5.5.1 Exercise: Visualizing Missing Data

Visualizing missing data is not as useless an endeavor as it might sound. There are informative ways of visualizing missingness in one variable if you have another variable that is associated with it. Let's introduce some missingness in our data so we have an example to play with:

Listing 5.12:

```
dat2 <- dat
nr <- nrow(dat2)
make.missing <- sample(seq(nr), floor(nr/10), prob = dat2$Response
    )
dat2$Response[ make.missing ] <- NA
```

Make a scatterplot of `Response` versus `Expsoure`, conditional on `Dose`, and add a "rug" on the `Exposure` axis that shows the `Exposure` values for which `Response` is missing. In other words, the result should look like this:



This figure suggests (correctly) that `Response` values are more likely to be missing at higher exposures, possibly a useful thing to recognize.

5.6 panel.superpose() and the panel.groups argument

One of the most important panel functions is called `panel.superpose`. When `xyplot()` has a `groups` argument, the default value for `panel` actually resolves to `panel.superpose`, and there is yet another argument, `panel.groups`, that resolves to `panel.xyplot`. In other words, when you do this:

Listing 5.13:

```
xyplot(Response ~ Exposure | Month,
       groups = Sex,
       data = dat
      )
```

You are implicitly doing this:

Listing 5.14:

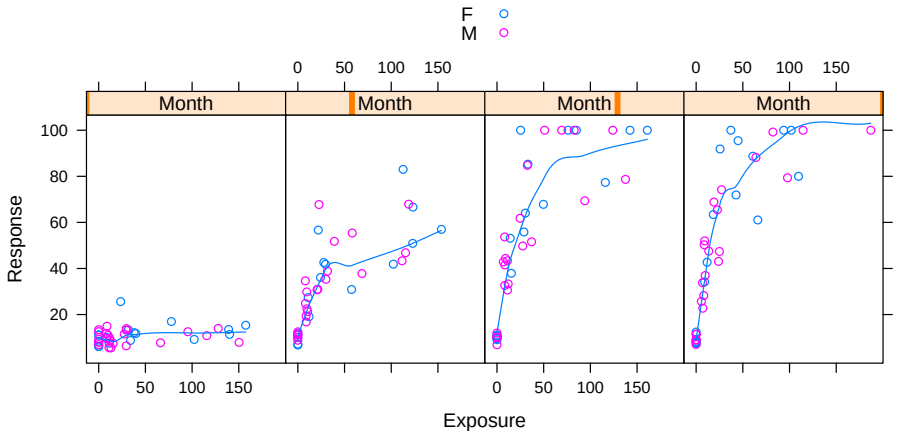
```
xyplot(Response ~ Exposure | Month,
       groups = Sex,
       data = dat,
       panel = panel.superpose,
       panel.groups = panel.xyplot
       )
```

When we specify `panel = panel.superpose`, we are essentially saying, "in each panel, for each level of the `groups` variable, go do whatever `panel.groups` tells you to do".

So, suppose we want to start drawing our loess fits again. If we want a single loess fit in each panel (based on combined data from both sexes), we would do:

Listing 5.15:

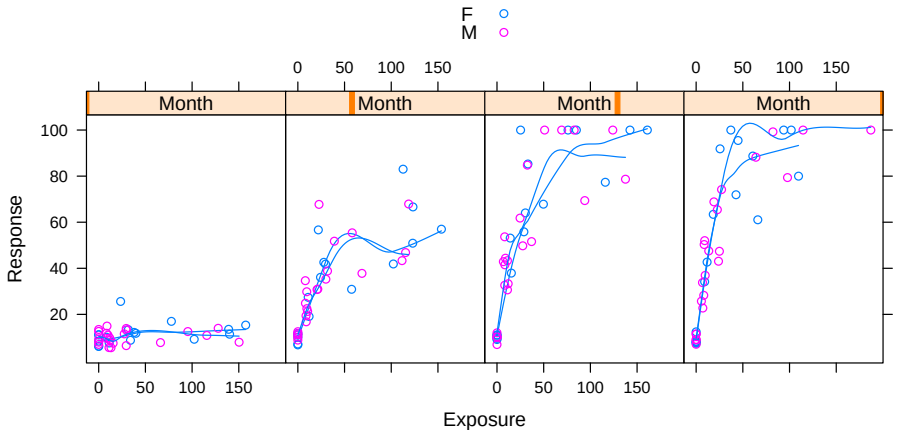
```
print(
xyplot(Response ~ Exposure | Month,
       groups = Sex,
       data = dat,
       panel = function(x, y, ...) {
         panel.superpose(x, y, ...)
         panel.loess(x, y)
         },
       panel.groups = panel.xyplot,
       layout = c(4, 1),
       auto.key = TRUE
       )
      )
```



However, if we want a separate loess fit for males and females within each panel, we would do:

Listing 5.16:

```
print(
xyplot(Response ~ Exposure | Month,
       groups = Sex,
       data = dat,
       panel = panel.superpose,
       panel.groups = function(x, y, ...) {
         panel.xyplot(x, y, ...)
         panel.loess(x, y)
       },
       layout = c(4, 1),
       auto.key = TRUE
       )
      )
```

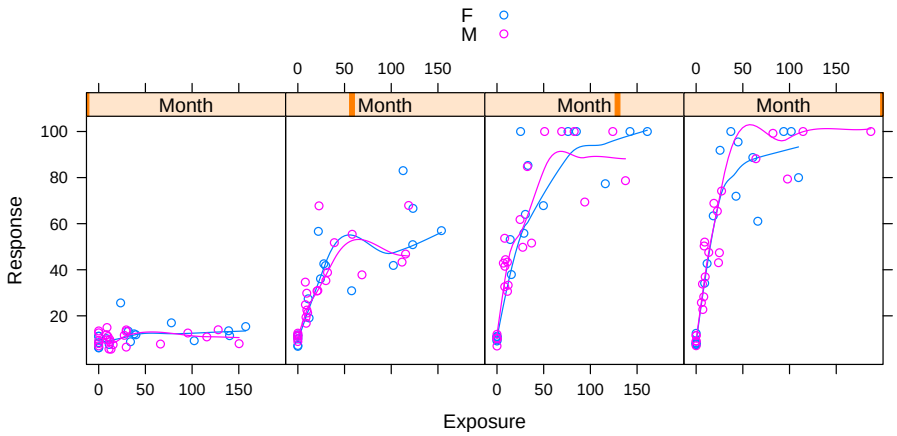


It's important to realize why the loess fits weren't drawn with different colors or line types: `xyplot()` was ready and willing to pass these plotting parameters to `panel.loess`, but our call to `panel.loess` didn't specify any arguments other than `x` and `y`. If we want "all that other stuff", we need to pass "`...`" to `panel.loess`, like this:

Listing 5.17:

```
print(
xyplot(Response ~ Exposure | Month,
       groups = Sex,
       data = dat,
       panel = panel.superpose,
       panel.groups = function(x, y, ...) {
         panel.xyplot(x, y, ...)
```

```
      panel.loess(x, y, ...)
    },
    layout = c(4, 1),
    auto.key = TRUE
    )
  )
```

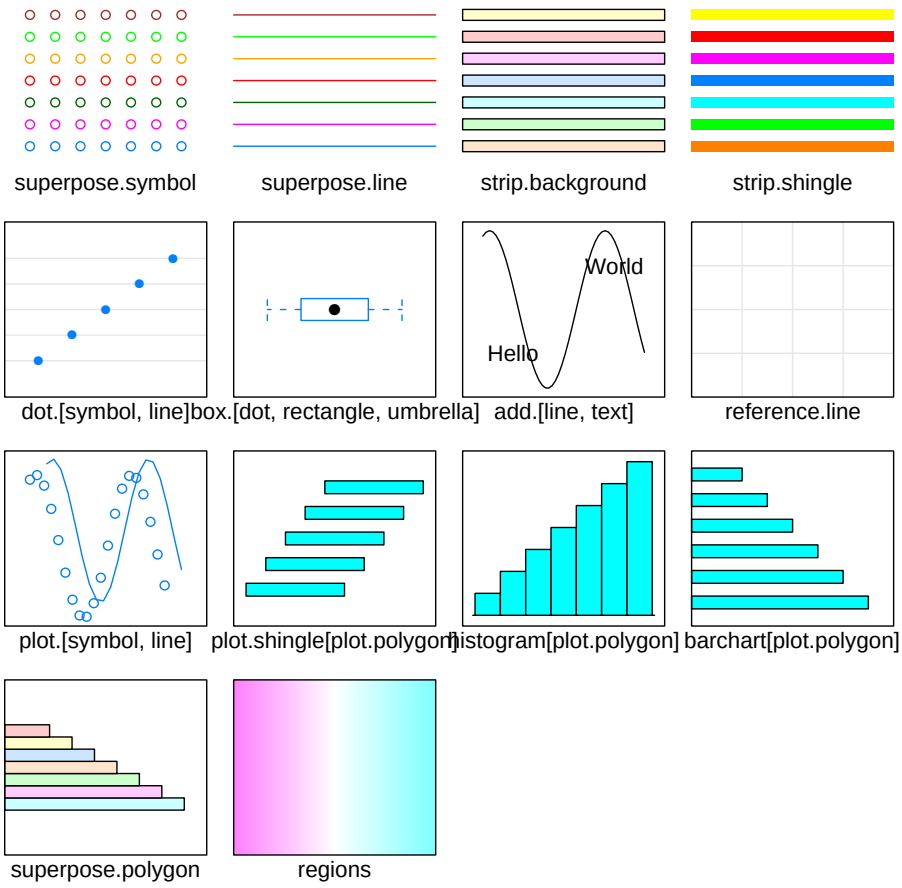


5.7 lattice themes

Let's suppose we are going to be printing out our graphics in black and white. Our most recent plot wouldn't work because we have distinguished the levels of `Sex` only by color. `xyplot()` made this choice for us, and it did so by consulting the current `theme`. To see many of the options that are governed by the `theme`, we can use the built-in lattice function `show.settings()`:

Listing 5.18:

```
show.settings()
```



For our current task, pay particular attention to the region of this figure that shows `superpose.symbol` and `superpose.line`. We can query the information that is represented here by doing:

Listing 5.19:

```
trellis.par.get('superpose.symbol')
```

```
$alpha
[1] 1 1 1 1 1 1 1

$cex
[1] 0.8 0.8 0.8 0.8 0.8 0.8 0.8

$col
[1] "#0080ff"    "#ff00ff"    "darkgreen" "#ff0000"
[5] "orange"     "#00ff00"    "brown"

$fill
```

```
[1] "#CCFFFF" "#FFCCFF" "#CCFFCC" "#FFE5CC" "#CCE6FF"
[6] "#FFFFCC" "#FFCCCC"

$font
[1] 1 1 1 1 1 1 1

$pch
[1] 1 1 1 1 1 1 1
```

Listing 5.20:

```
trellis.par.get('superpose.line')
```

```
$alpha
[1] 1

$col
[1] "#0080ff"   "#ff00ff"   "darkgreen" "#ff0000"
[5] "orange"    "#00ff00"   "brown"

$lty
[1] 1 1 1 1 1 1 1

$lwd
[1] 1 1 1 1 1 1 1
```

These are the parameters that `panel.superpose()` is going to cycle through as it goes through the different levels of the grouping variable. To change these parameters, we can do things like:

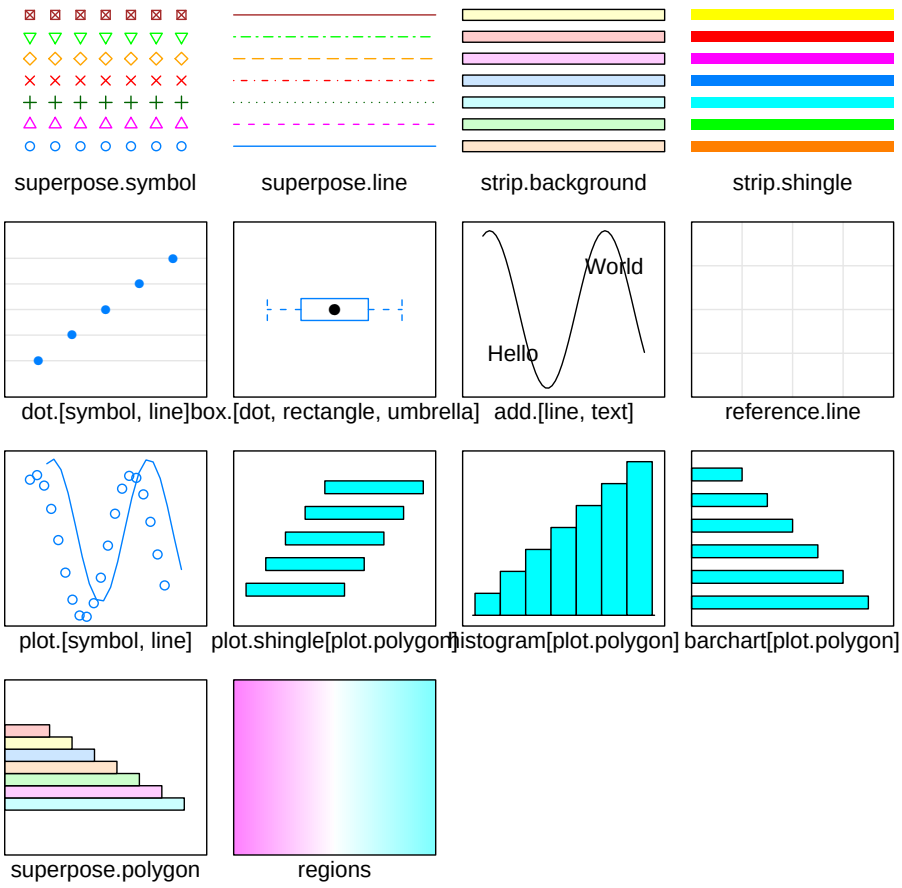
Listing 5.21:

```
trellis.par.set(superpose.symbol = list(pch = 1:7))
trellis.par.set(superpose.line = list(lty = 1:7))
```

Let's take a quick look at these choices before we use them:

Listing 5.22:

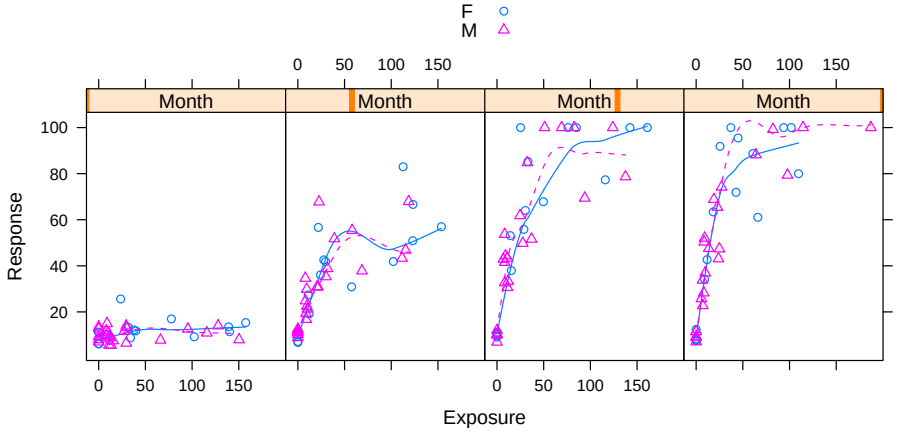
```
show.settings()
```



Now let's put these choices to work:

Listing 5.23:

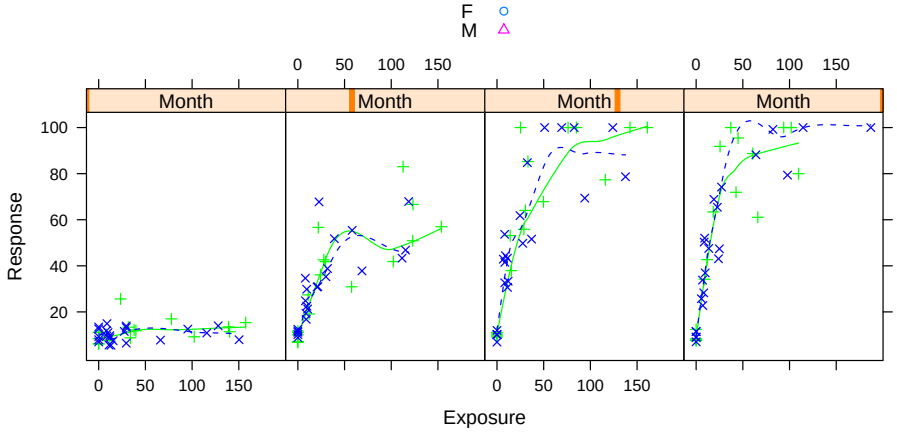
```
print(
xyplot(Response ~ Exposure | Month,
       groups = Sex,
       data = dat,
       panel = panel.superpose,
       panel.groups = function(x, y, ...) {
         panel.xyplot(x, y, ...)
         panel.loess(x, y, ...)
       },
       layout = c(4, 1),
       auto.key = TRUE
       )
      )
```



It is important to realize that the `theme` is also consulted when a key is created with `auto.key = TRUE`. Therefore, if you override the parameters specified in the `theme`, you should not expect `auto.key` to do the right thing. As an example of what **NOT** to do if you are using `auto.key`:

Listing 5.24:

```
print(
xyplot(Response ~ Exposure | Month,
       groups = Sex,
       data = dat,
       col = c("green", "blue"),
       pch = 3:4,
       panel = panel.superpose,
       panel.groups = function(x, y, ...) {
         panel.xyplot(x, y, ...)
         panel.loess(x, y, ...)
       },
       layout = c(4, 1),
       auto.key = TRUE
       )
      )
```



5.8 Homework

1. Plot `Response` versus `Month`, conditional on `Dose`. Use thin grey lines to connect data points from the same patient. Add a thick black line that connects the mean responses for each month.

2. Plot `Response` versus `Exposure`, conditional on `Month`, using `Dose` as the `groups` variable. Set the theme so that placebo is represented by blue circles and active doses are represented by red crosses that increase in size as the dose level increases.

Chapter 6

Data Assembly

6.1 Objectives

After completing this chapter, you will be able to . . .

- Reorder and rename data frame columns.
- Systematically transform column values.
- Uniquely identify rows by means of well-defined columns.
- Reorder data frame rows.
- Impute missing values column-wise or row-wise, or with constants.
- Generate new columns based on existing columns.
- Remove arbitrary columns and rows.
- Combine two tables that may differ in logical structure.

6.2 Introduction

Chapter 5 explained how to reduce data using summary techniques. This chapter explains somewhat the opposite: building data up by combining different sources. In Pharmacometrics, a common task is the assembly of a single analyis data set from various sources: pharmacokinetic measurements, demographic tables, dosing histories, laboratory analyses, concomitant medications, and so on. We'll cover techniques that allow us to assemble data flexibly, efficiently, and intelligibly.

6.3 Considerations

When faced with an assembly task, it is tempting to leap right in and start drafting a script. Time spent **planning**, however, will be rewarded. In particular, it is much to your advantage if you or someone else has prepared a specification identifying how the assembled data set should look, and whence the various components should derive. If available, a 'data spec' will reduce time wasted on irrelevant data, and time wasted 'fixing' a data set not suited for the intended analysis.

The reality is that, even when a data spec is available, it may change during the course of the project. In fact, even the data themselves (itself?) may change. All this **dynamism** is due in part to the pressure of deadlines. Work may begin before all the data is even collected, let alone 'cleaned' and 'locked'.

It takes flexibility to survive in a dynamic environment. The chief way to achieve flexibility (while maintaining efficiency and intelligibility) is to focus on the preparation of source data. Merging multiple sources is in some sense the focus of this chapter. However, more effort will be devoted to source preparation. If the sources are well-designed, they can be quickly rearranged into a variety of valid products to accommodate changing demands.

Another important consideration is one so unquestionably true that it almost sounds strange to mention it (but I'll do so anyway): modern data management is table-oriented. The ubiquity of **table orientation** is no surprise. Tables are great for cataloging attributes of things, perhaps because their two-dimensional structure fits well on square, flat interfaces like paper and computer screens. 'Two' is a convenient number of dimensions: we can let various columns represent multiple attributes of an object, (subject name, subject height, subject weight) and can let rows represent multiple similar things (e.g., all the subjects in a study). One dimension put to the same purpose would be inefficient, and three dimensions is often more than our minds or interfaces can handle conveniently.

However, table-oriented data management suffers from one critical drawback: most information is not two-dimensional. For example, if we wish to tabluate 'dosing history' next to 'subject name' and 'subject height', we quickly realize that the 'one-row-per-subject' scheme is incompatible: we need to repeat some identifying information on all those history rows. And if we want to to have separate history rows for doses and PK samples, we soon notice that the 'sample concentration' column doesn't have a natural meaning for the dose rows, and the 'dose amount' column doesn't have a natural meaning for the sample rows. The principal challenges of data assembly arise precisely from the constraint of representing poly-dimensional data in a two-dimensional form.

6.4 Source Management

The outline of the rest of the chapter is a general recommendation for a data assembly sequence. Generally, one should put significant effort into preparing the source data sets, one at a time, and then proceed to merge them. Source management generally starts with column changes and row changes, followed by imputation of missing data and generation of new variables. Next, fix any errors, drop unnecessary rows and columns, and reorganize if necessary.

The order of activities is not fixed. For example, it may be necessary to generate a variable to serve as a source for imputation. It may be necessary to impute some missing values before the rows can be correctly ordered. Maybe you want to drop some rows and columns immediately. It may be necessary to reorganize your data before processing it, rather than after. Do whatever promotes efficiency and clarity.

Before managing source data, you will of course have to "read it in". The `as.is` argument to many of the `read` functions is useful, since it assures that columns looking like text will behave like text (rather than factors). If your underlying data is still changing, you may have to do some version management as well, perhaps external to the assembly process.

6.4.1 Column Management

Columns are the main elements of a table. Indeed, a data.frame is really a list of equal-length columns. Recall that all the elements of a column share a single data type.

Reorder

Your specification (whether real or implicit) may call for a different column order than that of the stored source. In R, columns are easily reordered by name or number, using the subset operators. Note that you can drop or even repeat columns during the same operation, so this could be a good time to get rid of unneeded information.

Listing 6.1:

```
head(Theoph)
```

```
  Subject   Wt Dose Time  conc
1       1 79.6 4.02 0.00  0.74
2       1 79.6 4.02 0.25  2.84
3       1 79.6 4.02 0.57  6.57
4       1 79.6 4.02 1.12 10.50
5       1 79.6 4.02 2.02  9.66
6       1 79.6 4.02 3.82  8.58
```

Listing 6.2:

```
Theoph <- Theoph[,c('Subject','Time','Wt','conc')]
head(Theoph)
```

```
  Subject Time   Wt  conc
1       1 0.00 79.6  0.74
2       1 0.25 79.6  2.84
3       1 0.57 79.6  6.57
4       1 1.12 79.6 10.50
5       1 2.02 79.6  9.66
6       1 3.82 79.6  8.58
```

Note that `Dose` was droped and `Time` moved next to `Subject`.

Rename

One of the safest ways to rename a column is like this.

Listing 6.3:

```
names(Theoph)[names(Theoph)=='Wt'] <- 'Weight'
head(Theoph)
```

```
  Subject Time Weight  conc
1       1 0.00   79.6  0.74
2       1 0.25   79.6  2.84
3       1 0.57   79.6  6.57
4       1 1.12   79.6 10.50
5       1 2.02   79.6  9.66
6       1 3.82   79.6  8.58
```

Type Coercion

If an otherwise-numeric column has some stray text, R makes the whole thing `character` when reading. You may need to coerce to numeric.

Listing 6.4:

```
Theoph$conc[[2]] <- 'BLQ'
class(Theoph$conc)
```

```
[1] "character"
```

Listing 6.5:

```
Theoph$conc <- as.numeric(Theoph$conc)
head(Theoph)
```

```
  Subject Time Weight  conc
1       1 0.00   79.6  0.74
2       1 0.25   79.6    NA
3       1 0.57   79.6  6.57
4       1 1.12   79.6 10.50
5       1 2.02   79.6  9.66
6       1 3.82   79.6  8.58
```

Next, consider this:

Listing 6.6:

```
unique(Theoph$Subject)
```

```
 [1] 1  2  3  4  5  6  7  8  9  10 11 12
12 Levels: 6 < 7 < 8 < 11 < 3 < 2 < 4 < 9 < 12 < ... < 5
```

Listing 6.7:

```
unique(as.numeric(Theoph$Subject))
```

```
 [1] 11  6  5  7 12  1  2  3  8 10  4  9
```

Surprised? Theoph$Subject is a (strangely) ordered factor. It is often preferable to coerce factors to text before doing other manipulations:

Listing 6.8:

```
Theoph$Subject <- as.character(Theoph$Subject)
unique(Theoph$Subject)
```

```
 [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10" "11"
[12] "12"
```

Listing 6.9:

```
unique(as.numeric(Theoph$Subject))
```

```
 [1]  1  2  3  4  5  6  7  8  9 10 11 12
```

That result was easier to anticipate than listing 7.

Value Coercion

Sometimes we want to leave the column type untouched, but change all the values systematically.

For `factor`, we can reassign the levels. (Reverting to the original `Theoph` ....)

Listing 6.10:

```
levels(Theoph$Subject)
```

```
 [1] "6"  "7"  "8"  "11" "3"  "2"  "4"  "9"  "12" "10" "1"
[12] "5"
```

Listing 6.11:

```
levels(Theoph$Subject) <- c(letters[1:12])
head(Theoph[!duplicated(Theoph$Subject),])
```

```
   Subject   Wt Dose Time conc
1        k 79.6 4.02    0 0.74
12       f 72.4 4.40    0 0.00
23       e 70.5 4.53    0 0.00
34       g 72.7 4.40    0 0.00
45       l 54.6 5.86    0 0.00
56       a 80.0 4.00    0 0.00
```

For `character`, convert temporarily to factor.

Listing 6.12:

```
Theoph$race <- rep(
       c(
              'white','black','asian','hispanic',
              'white','black','asian','hispanic',
              'white','black','asian','hispanic'
       ),
       each=11
)
head(Theoph[!duplicated(Theoph$Subject),])
```

```
   Subject   Wt Dose Time conc     race
1        k 79.6 4.02    0 0.74    white
12       f 72.4 4.40    0 0.00    black
23       e 70.5 4.53    0 0.00    asian
34       g 72.7 4.40    0 0.00 hispanic
45       l 54.6 5.86    0 0.00    white
56       a 80.0 4.00    0 0.00    black
```

Listing 6.13:

```
Theoph$race <- as.character(
       factor(
              Theoph$race,
              levels=c('white','black','asian','hispanic'),
              labels=c('wh','bl','as','hi')
       )
)
head(Theoph[!duplicated(Theoph$Subject),])
```

```
   Subject   Wt Dose Time conc race
1        k 79.6 4.02    0 0.74   wh
12       f 72.4 4.40    0 0.00   bl
23       e 70.5 4.53    0 0.00   as
34       g 72.7 4.40    0 0.00   hi
45       l 54.6 5.86    0 0.00   wh
56       a 80.0 4.00    0 0.00   bl
```

For `numeric`, we may need to convert units, e.g. from mg/L to μg/L.

Listing 6.14:

```
Theoph$conc <- Theoph$conc * 1000
head(Theoph)
```

```
  Subject   Wt Dose Time  conc race
1       k 79.6 4.02 0.00   740   wh
2       k 79.6 4.02 0.25  2840   wh
3       k 79.6 4.02 0.57  6570   wh
4       k 79.6 4.02 1.12 10500   wh
5       k 79.6 4.02 2.02  9660   wh
6       k 79.6 4.02 3.82  8580   wh
```

The base function `within()` achieves the same thing but is perhaps easier to read.

Listing 6.15:

```
Theoph <- within(Theoph, conc <- conc * 1000)
```

Dates and times are often problematic. They look like text but need to behave as quantities. Here, we'll use `strptime` to make a date-time object. Do see the extensive formatting options in the help file. For a convenient interface to `strptime`, see `?temporal` in the package `metrumrg`.

Listing 6.16:

```
strptime('Thu 19 Mar 09 03:48',format='%a %d %b %y %H:%M')
strptime('3/19/2009 03.48',    format='%m/%d/%Y %H.%M')
```

```
[1] "2009-03-19 03:48:00"
```

```
[1] "2009-03-19 03:48:00"
```

6.4.2 Row Management

Row management generally concerns details that are determined on a per-row basis. That said, column information is still fairly important for managing rows.

Names

Just like columns, every row in a data frame has a name. Row names, however, must be unique. (It's a good idea for column names to be unique, too, but this is not enforced.) After a series of transformations, rownames can get pretty ugly. You can always view them with `rownames()` or `row.names()`. In later versions of R, you can reset them by executing `rownames(x)<- NULL`. Row names are never missing, and are always character, even if they look like numbers. (They are *names* after all.)

Keys

Even though rows are *formally* distinguishable by their official names, they should be *structurally* distinguishable by data frame content. That's where the concept of **keys** comes into play. Every row in your data frame should be there for a special reason; it should do something that no other row can do; it should have a unique purpose. **A key is a set of columns that uniquely distinguishes every row.** (Data theorists would maybe call this a 'unique key', but let's not quibble.) That is, for some set of columns (maybe only one), each row-level combination of values should be unique. If you can't say this for a particular data frame, you should seriously ask yourself what the purpose of the data frame is.

It's easy enough to meet the uniqueness requirement using random data. But that doesn't make a useful key. The key should contain information that participates in the symantics of your project. For example, a demographics data frame may be keyed on `Subject`: that is, exactly one row for every subject in the study. Dose history may be keyed on `Subject` and `Date`, assuming no more than one dose per subject per date. An ECG data frame may be keyed on `Subject`, `Date`, `Time`, and `Replicate` (assuming simultaneous replicates).

Avoid superkeys. A **superkey** uniquely distinguishes every row using more columns than necessary. If you consider *all* a table's columns to be the key, you will almost certainly be able to distinguish all rows, but that would be pointless. In general, the key columns

should *identify* unique objects (persons, events) and the other columns should *characterize* (list attributes of) those objects.

Many data management systems explicitly recognize the concept of keys. R does not. This means that maintaining keys is up to you, because there are really no programming constructs to help you (at least in the base packages). Just make sure that your keys are **definite** (no NAs) and **unique** (no duplicates), in that order. If you are missing a key field for a particular row, there will likely be problems later. If the value can't be recovered, probably the row is not useful. What good, for instance, is a full row of demographic information if you don't know who the subject was? Similarly, a PK record is of limited value if you have no idea when the sample was taken. Also, if you have two PK records for the same subject on the same date at the same time, you need to know why. Is the subject mis-identified? Is the date wrong? Are these replicate samples? You may need to drop or aggregate some records before proceeding.

Order

In classical data management theory, row order shouldn't matter. However, a table with ordered rows is easier to read. More importantly, row order can simplify many calculations. "Last observation carried forward" is rather trivial for rows sorted on 'time', but could get fairly complex for randomly ordered rows.

In nearly all cases, the best row order is that based on the key. Use `order()` to construct a permutation of row order based on the key, and pass that as the row argument to the subset operator. (Reverting to the original `Theoph` ....)

Listing 6.17:

```
sorted <- Theoph[order(Theoph$Subject, Theoph$Time),]
identical(sorted,Theoph)
```

```
[1] FALSE
```

Listing 6.18:

```
head(sorted)
```

```
   Subject Wt Dose Time conc
56       6 80    4 0.00 0.00
57       6 80    4 0.27 1.29
58       6 80    4 0.58 3.08
59       6 80    4 1.15 6.44
60       6 80    4 2.03 6.32
61       6 80    4 3.57 5.53
```

Listing 6.19:

```
head(Theoph)
```

```
  Subject   Wt Dose Time  conc
1       1 79.6 4.02 0.00  0.74
2       1 79.6 4.02 0.25  2.84
3       1 79.6 4.02 0.57  6.57
4       1 79.6 4.02 1.12 10.50
5       1 79.6 4.02 2.02  9.66
6       1 79.6 4.02 3.82  8.58
```

6.4.3 Imputations

At this point in the assembly process, we know that our columns are appropriately named and typed, and all our rows are uniquely identifiable and sorted. Now is a good time to impute any missing values, if necessary. Make sure missing values are actually stored as the special value `NA` (not character `'missing'`, `'unknown'`,`""`, etc.). Then decide which type of imputation is appropriate.

Constant

For a constant imputation, all missing values in a vector receive the same value. (This trivial example actually does nothing.)

Listing 6.20:

```
Theoph$Wt[is.na(Theoph$Wt)] <- mean(Theoph$Wt,na.rm=TRUE)
```

Horizontal

For a horizontal imputation, all the information needed for each NA is contained in the same row. (Using some invented data ....)

Listing 6.21:

```
pk <- data.frame(
       subject=rep(letters[1:3],each=3),
       time=rep(1:3,3),
       baseline=rep(70:72,each=3),
       weight=c(70,70.5,69,71,NA,70,72, 72,NA)
)
pk
```

```
  subject time baseline weight
1       a    1       70   70.0
2       a    2       70   70.5
3       a    3       70   69.0
4       b    1       71   71.0
5       b    2       71     NA
6       b    3       71   70.0
7       c    1       72   72.0
8       c    2       72   72.0
9       c    3       72     NA
```

Listing 6.22:

```
pk$weight[is.na(pk$weight)] <- pk$baseline[is.na(pk$weight)]
pk
```

```
  subject time baseline weight
1       a    1       70   70.0
2       a    2       70   70.5
3       a    3       70   69.0
4       b    1       71   71.0
5       b    2       71   71.0
6       b    3       71   70.0
7       c    1       72   72.0
8       c    2       72   72.0
9       c    3       72   72.0
```

Vertical

For a vertical imputation, all the information needed for each NA is contained in the same column. "Last observation carried forward" is a good example. We'll use `locf` from the `metrumrg` package.

Listing 6.23:

```
pk
```

```
  subject time weight
1       a    1   70.0
2       a    2   70.5
3       a    3   69.0
4       b    1   71.0
5       b    2     NA
6       b    3   70.0
7       c    1   72.0
```

```
8       c    2   72.0
9       c    3     NA
```

Listing 6.24:

```
library(metrumrg)
```

Listing 6.25:

```
pk$weight <- locf(pk$weight)
pk
```

```
  subject time weight
1       a    1   70.0
2       a    2   70.5
3       a    3   69.0
4       b    1   71.0
5       b    2   71.0
6       b    3   70.0
7       c    1   72.0
8       c    2   72.0
9       c    3   72.0
```

Stratified

Suppose, however, that one of our subjects is missing the first weight observation.

Listing 6.26:

```
pk
```

```
  subject time weight
1       a    1   70.0
2       a    2   70.5
3       a    3   69.0
4       b    1     NA
5       b    2   71.0
6       b    3   70.0
7       c    1   72.0
8       c    2   72.0
9       c    3     NA
```

Listing 6.27:

```
pk$weight <- locf(pk$weight)
pk
```

```
  subject time weight
1       a    1   70.0
2       a    2   70.5
3       a    3   69.0
4       b    1   69.0
5       b    2   71.0
6       b    3   70.0
7       c    1   72.0
8       c    2   72.0
9       c    3   72.0
```

Subject `b` now has a weight imputed using data from subject `a`! This is Very Bad. Vertical imputations can be dangerous: they do not necessarily respect subset boundaries.

What we need here is a stratified imputation. Somehow we need to conduct the imputation independently for subsets of the data. One approach is to break the data into subsets, conduct the imputation on each set, and reassemble. It is easier, though, to use the function `reapply`, discussed later in this chapter.

6.4.4 Derived Variables

Our columns are formatted, our rows are organized and we've imputed as many missings as possible. But the specification calls for some of the data to be presented in a different form. The tools and techniques we need depend on the type of derivation.

Extractions

An extraction is a piece of text derived from some other column. (Make sure that 'other column' is not a factor!) Suppose you have a universal subject identifier that includes protocol information. You need a column with just the protocol number. If the column is fixed-width, you can use the substring function.

Listing 6.28:

```
substr('PROT123-SITE001-SUBJ200', start=5,stop=7)
```

```
[1] "123"
```

If the the source column has a format that is more variable, you can often achieve satisfying results using regular expressions. For more information, see `regex()`, `sub()`, `gsub()`, and `strsplit()`. Regular expressions constitute an advanced topic and will not be further dicussed here.

Combinations

The reverse of an extraction, a combination puts together fragments of text from other columns. Use `paste()`. For example, you could create a universal subject identifier by pasting together Protocol, Site, and Subject columns. The arguments to `paste()` do not need to be character, but they will be coerced to character.

Calculations

The `Dose` column in `Theoph` is stated in mg/kg. We can calculate the absolute dose from dose and weight (well, really: mass).

Listing 6.29:

```
Theoph$Amount <- Theoph$Dose * Theoph$Wt
head(Theoph)
```

```
  Subject   Wt Dose Time  conc  Amount
1       1 79.6 4.02 0.00  0.74 319.992
2       1 79.6 4.02 0.25  2.84 319.992
3       1 79.6 4.02 0.57  6.57 319.992
4       1 79.6 4.02 1.12 10.50 319.992
5       1 79.6 4.02 2.02  9.66 319.992
6       1 79.6 4.02 3.82  8.58 319.992
```

The base function `with()` is easier on the eyes, especially for complicated expressions.

Listing 6.30:

```
Theoph$Amount <- with(Theoph, Dose * Wt)
```

You may find `transform()` an even more elegant alternative.

Listing 6.31:

```
Theoph <- transform(Theoph, Amount = Dose * Wt)
head(Theoph)
```

```
  Subject   Wt Dose Time  conc  Amount
1       1 79.6 4.02 0.00  0.74 319.992
2       1 79.6 4.02 0.25  2.84 319.992
3       1 79.6 4.02 0.57  6.57 319.992
4       1 79.6 4.02 1.12 10.50 319.992
5       1 79.6 4.02 2.02  9.66 319.992
6       1 79.6 4.02 3.82  8.58 319.992
```

Aggregations

Suppose we want a new column in Theoph representing the maximum concentration per subject (C_{max}). This is really a two-step operation. First, we need to aggregate concentration by subject, searching for the maximum. Then we need to merge that information with the existing data frame. The function `tapply()` (Chapter 5) will do the aggregating for us, but we haven't discussed merging yet. `reapply()` (metrumrg package) is a wrapper for `tapply()` that stretches the result to fit the original data: effectively combining both steps.

Listing 6.32:

```
args(reapply)
```

```
function (x, INDEX, FUN, ...)
NULL
```

Now we calculate C_{max}. (Reverting to the original `Theoph` ....)

Listing 6.33:

```
Theoph$Cmax <- with(
       Theoph,
       reapply(
               conc,
               INDEX=Subject,
               FUN=max
       )
)
head(Theoph)
```

```
  Subject   Wt Dose Time  conc Cmax
1       1 79.6 4.02 0.00  0.74 10.5
2       1 79.6 4.02 0.25  2.84 10.5
3       1 79.6 4.02 0.57  6.57 10.5
4       1 79.6 4.02 1.12 10.50 10.5
5       1 79.6 4.02 2.02  9.66 10.5
6       1 79.6 4.02 3.82  8.58 10.5
```

Flags

Typically, data assembly requires creation of indicator variables. Usually these are binary fields, telling whether a given condition applies. We'll call these 'flags'. In R they would normally be of type `logical`, but ones and zeros are much easier to read than `FALSE`s and `TRUE`s, especially for data sets with many flags.

For example, let's flag all the rows in `Theoph` where concentration is maximum.

Listing 6.34:

```
Theoph <- transform(Theoph, Tmax = as.integer(conc == Cmax))
head(Theoph)
```

```
  Subject   Wt Dose Time  conc Cmax Tmax
1       1 79.6 4.02 0.00  0.74 10.5    0
2       1 79.6 4.02 0.25  2.84 10.5    0
3       1 79.6 4.02 0.57  6.57 10.5    0
4       1 79.6 4.02 1.12 10.50 10.5    1
5       1 79.6 4.02 2.02  9.66 10.5    0
6       1 79.6 4.02 3.82  8.58 10.5    0
```

Now we can get some idea when T_{max} is occuring.

Listing 6.35:

```
Theoph$Time[as.logical(Theoph$Tmax)]
```

Listing 6.36:

```
#Alternatively....
with(Theoph, Time[!!Tmax])
```

```
 [1] 1.12 1.92
1.02 1.07 1.00 1.15 3.48 2.02 0.63 3.55 0.98
[12] 3.52
```

The construct `!!` is shorthand for `as.logical()`. The `!` operator forces R to coerce the zeros and ones to logical. The second `!` restores the reversed logic.

6.4.5 Cell Management

At some point in the assembly process you may discover cell values that are in error. If they are few, you may wish to correct them in your script. It is probably better to start with a correct data source: this is really a version management issue. If that is not possible, and if errors are many, consider tabulating them externally in a secondary source, and writing code that applies your QC findings systematically.

Altering the values of cells is usually performed by the assignment version of the subset operator: `[<-`.

Listing 6.37:

```
Theoph$conc[with(Theoph, Subject==1 & Time==0)]
```

```
[1] 0.74
```

Listing 6.38:

```
Theoph$conc[with(Theoph, Subject==1 & Time==0)] <- 0.85
Theoph$conc[with(Theoph, Subject==1 & Time==0)]
```

```
[1] 0.85
```

It is possible, of course, to identify a particular value by postion:

Listing 6.39:

```
Theoph$conc[1,5] <- 0.85
```

However, it is safer to test for values of the key. Positions may change without notice in a future version of the data.

6.4.6 Restrictions

It's time to drop unnecessary rows and columns, if you haven't done so already. An alternative to dropping rows is to flag them somehow to prevent their inclusion in later analyses. The resulting data set is more informative, and still usable according to the intended restrictions.

Uninformative columns can often be dropped. For example, if a Dose column gives essentially the same information as a Treatment Group column, perhaps only one is needed. Especially where one column contains only a subset of the data defined in a similar column, prefer the more complete instance. Some of these decisions may need to be made much earlier in the assembly process.

Dropping noninformative records should, of course, be sensitive to the pharmacokinetic context. Dropping an unknown (`NA`) value for dose is likely to be far more consequential than dropping an unknown sample concentration. Perhaps the subject's remaining history should be ignored, if the dose datum cannot be resolved.

Rows and columns can be dropped implicitly by subsetting the data frame to the dimensions not being dropped. The most direct way to drop a row is to subset with the negative index of the row in question.

Listing 6.40:

```
head(Theoph)
```

```
  Subject   Wt Dose Time  conc Cmax Tmax
1       1 79.6 4.02 0.00  0.85 10.5    0
2       1 79.6 4.02 0.25  2.84 10.5    0
3       1 79.6 4.02 0.57  6.57 10.5    0
4       1 79.6 4.02 1.12 10.50 10.5    1
5       1 79.6 4.02 2.02  9.66 10.5    0
6       1 79.6 4.02 3.82  8.58 10.5    0
```

Listing 6.41:

```
Theoph <- Theoph[-1,]
head(Theoph)
```

```
  Subject   Wt Dose Time  conc Cmax Tmax
2       1 79.6 4.02 0.25  2.84 10.5    0
3       1 79.6 4.02 0.57  6.57 10.5    0
4       1 79.6 4.02 1.12 10.50 10.5    1
5       1 79.6 4.02 2.02  9.66 10.5    0
6       1 79.6 4.02 3.82  8.58 10.5    0
7       1 79.6 4.02 5.10  8.36 10.5    0
```

That technique works for columns, too. But you can also set a column to NULL, which is perhaps even more direct.

Listing 6.42:

```
Theoph$Wt <- NULL
head(Theoph)
```

```
  Subject Dose Time  conc Cmax Tmax
2       1 4.02 0.25  2.84 10.5    0
3       1 4.02 0.57  6.57 10.5    0
4       1 4.02 1.12 10.50 10.5    1
5       1 4.02 2.02  9.66 10.5    0
6       1 4.02 3.82  8.58 10.5    0
7       1 4.02 5.10  8.36 10.5    0
```

6.4.7 Reorganization

Source data may require rearrangement to be compatible with anticipated output formats. Suppose you have measurements arranged by occasion. You may need to rearrange to a 'tall-skinny' layout (mentioned in Basic Plotting).

Listing 6.43:

```
days
```

```
  subject  day1  day2   day3
1       a  0.24 -1.60  0.550
2       b  0.29  0.37 -0.075
3       c -0.36 -0.98  0.710
4       d -0.84 -0.35  0.260
```

Listing 6.44:

```
tall <- stack(days)
tall$subject <- days$subject
tall
```

```
   values  ind subject
1   0.240 day1       a
2   0.290 day1       b
3  -0.360 day1       c
4  -0.840 day1       d
5  -1.600 day2       a
6   0.370 day2       b
7  -0.980 day2       c
8  -0.350 day2       d
9   0.550 day3       a
10 -0.075 day3       b
11  0.710 day3       c
12  0.260 day3       d
```

Use unstack to acheive the reverse.

Listing 6.45:

```
labs
```

```
   subject       test  value
1        a     parent  0.540
2        b     parent -1.000
3        c     parent -0.240
4        d     parent  0.840
5        a metabolite -0.220
6        b metabolite -0.041
7        c metabolite -0.660
8        d metabolite -0.930
9        a   caffeine  0.710
10       b   caffeine  1.000
11       c   caffeine  0.390
12       d   caffeine  0.720
```

Listing 6.46:

```
short <- unstack(labs,form=value ~ test)
short$subject <- unique(labs$subject)
short
```

```
  parent metabolite caffeine subject
1   0.54     -0.220     0.71       a
```

```
2  -1.00     -0.041     1.00       b
3  -0.24     -0.660     0.39       c
4   0.84     -0.930     0.72       d
```

Caution: the `unstack` example makes dangerous assumptions about the organization of the data. Try unstacking `subject`, too, and make sure columns agree.

6.5 Merge Management

Merging is more related to the concept 'assembly' than anything we've discussed so far. Merging combines two data frames. In relational database theory, it's called 'joining'. In R, a merge is typically accomplished by the function with the same name.

6.5.1 Order

Only two tables can be merged at a time. Typically, we have more than two source tables in a project, which must therefore be merged sequentially. Usually one should proceed from greatest 'key scope' to least. For example, first merge tables that are keyed on Subject, Date, and Time; then add in those keyed on Subject and Date; and finally those keyed on Subject. Thus, the more general data (simpler key) will be included on all rows where it applies.

6.5.2 Technique

We'll create some sample data tables to merge.

Listing 6.47:

```
dose <- data.frame(
       subject = rep(letters[1:3], each = 2),
       time = rep(c(1,3),3),
       amount = rep(c(40,60,80), each = 2)
)
pk <- data.frame(
       subject = rep(letters[1:3], each = 4),
       time = rep(1:4,3),
       conc = signif(rnorm(12),2) + 2
)
demo <- data.frame(
       subject = letters[1:4],
       race = c('asian','white','black','other'),
```

```
        sex = c('female','male','female','male'),
        weight = c(75, 70, 73, 68)
)
```

Listing 6.48:

```
dose
```

```
  subject time amount
1       a    1     40
2       a    3     40
3       b    1     60
4       b    3     60
5       c    1     80
6       c    3     80
```

Listing 6.49:

```
pk
```

```
   subject time  conc
1        a    1 2.110
2        a    2 1.900
3        a    3 2.400
4        a    4 2.520
5        b    1 2.700
6        b    2 3.100
7        b    3 0.200
8        b    4 0.900
9        c    1 1.917
10       c    2 1.500
11       c    3 1.490
12       c    4 1.000
```

Listing 6.50:

```
demo
```

```
  subject  race    sex weight
1       a asian female     75
2       b white   male     70
3       c black female     73
4       d other   male     68
```

The `merge()` function combines data frames by examining their keys. You can specify the keys using the `by` argument. By default, `merge()` will use whatever columns the two data frames have in common. As the help file says, "The rows in the two data frames that match on the specified columns are extracted, and joined together".

Listing 6.51:

```
merge(dose,pk)
```

```
  subject time amount  conc
1       a    1     40 2.110
2       a    3     40 2.400
3       b    1     60 2.700
4       b    3     60 0.200
5       c    1     80 1.917
6       c    3     80 1.490
```

Note that a lot of the pk data is missing. That's because there were no doses at those times. But we want all the pk data regardless, so the following is better.

Listing 6.52:

```
merge(dose, pk, all=TRUE)
```

```
   subject time amount  conc
1        a    1     40 2.110
2        a    2     NA 1.900
3        a    3     40 2.400
4        a    4     NA 2.520
5        b    1     60 2.700
6        b    2     NA 3.100
7        b    3     60 0.200
8        b    4     NA 0.900
9        c    1     80 1.917
10       c    2     NA 1.500
11       c    3     80 1.490
12       c    4     NA 1.000
```

Note the `NA` amounts. They are structural missings, not logical missings. They arise not from any deficiency in the data, but from the limitation of representing poly-dimensional data in a two-dimensional form (Section 3).

Next, we'll apply the demographic data. Apparently subject `d` never actually participated in the study, so we'll quietly igore by not saying `all=TRUE`.

Listing 6.53:

```
merge(
       merge(dose, pk, all=TRUE), #like before
       demo                       #new data
)
```

```
   subject time amount  conc  race    sex weight
1        a    1     40 2.110 asian female     75
2        a    2     NA 1.900 asian female     75
3        a    3     40 2.400 asian female     75
4        a    4     NA 2.520 asian female     75
5        b    1     60 2.700 white   male     70
6        b    2     NA 3.100 white   male     70
7        b    3     60 0.200 white   male     70
8        b    4     NA 0.900 white   male     70
9        c    1     80 1.917 black female     73
10       c    2     NA 1.500 black female     73
11       c    3     80 1.490 black female     73
12       c    4     NA 1.000 black female     73
```

The `merge()` function is a good way to combine data frames that have related data but different structures. But there are a few other techniques that are conceptually equivalent. If two data frames have the same number of rows, and corresponding keys, then `cbind()` is a more efficient subsitute for `merge()`. Similarly, if two data frames have the same number of columns, with corresponding meanings and types, then `rbind()` is a more efficient substitute for `merge()`. In addition to being more efficient, `rbind()` and `cbind()` accept more than two arguments (tables) at a time. The function `reapply()` discussed earlier is essentially an aggregate step followed by a one-column merge.

The merge of demographic data above was a little risky. Subject `d` was in the demographic data but not in the clinical data. What if someone had been represented in the clinical data, but not in the demographic data?

Listing 6.54:

```
merge(
       merge(dose, pk, all=TRUE), #like before
       demo[-1,]                  #ignore the first subject
)
```

```
  subject time amount  conc  race    sex weight
1       b    1     60 2.700 white   male     70
2       b    2     NA 3.100 white   male     70
3       b    3     60 0.200 white   male     70
4       b    4     NA 0.900 white   male     70
5       c    1     80 1.917 black female     73
6       c    2     NA 1.500 black female     73
7       c    3     80 1.490 black female     73
8       c    4     NA 1.000 black female     73
```

Subject `a`'s clinical data has been silently dropped! While we don't necessarily want to see all the demographic data, we do want to see all the dosing and pk data. `merge()` lets us

specify `all` independently for each data frame.

Listing 6.55:

```
merge(
       merge(dose, pk, all=TRUE), #like before
       demo[-1,],                 #ignore the first subject
       all.x=TRUE                 #show all of x regardless
)
```

```
   subject time amount  conc  race    sex weight
1        a    1     40 2.110  <NA>   <NA>     NA
2        a    2     NA 1.900  <NA>   <NA>     NA
3        a    3     40 2.400  <NA>   <NA>     NA
4        a    4     NA 2.520  <NA>   <NA>     NA
5        b    1     60 2.700 white   male     70
6        b    2     NA 3.100 white   male     70
7        b    3     60 0.200 white   male     70
8        b    4     NA 0.900 white   male     70
9        c    1     80 1.917 black female     73
10       c    2     NA 1.500 black female     73
11       c    3     80 1.490 black female     73
12       c    4     NA 1.000 black female     73
```

That's called a left join. The function `stableMerge()` in the metrumrg package simplifies the left join, does some additional error checking, and guarrantees that rows and columns will not be reordered.

Listing 6.56:

```
stableMerge(
       merge(dose, pk, all=TRUE),
       demo[-1,]
)
```

Results are like those above.

6.5.3 Contextual Alterations

Earlier we discussed at length various imputations, derivations, and restrictions for source data. Some alterations aren't meaningful until after the merge: they are sensitive to context of the complete data. Now is the right time to address. For example, a variable like 'time after dose' can't be calculated for pk records until the dose records are present. Similarly, we may want to flag dose records that occur after the last pk record, but again, we need context. Don't be tempted to leave all the alterations for the post-merge workup, however. To paraphrase Einstein, every alteration should be made as soon as possible, but not sooner!

6.5.4 Characterization

When data assembly is essentially complete, it's a good idea to do some exploratory analysis. Some simple diagnostic plots may detect some anomalies that should be addressed before analysis.

6.6 Exercises

Enter `library(metrumrg)`. Enter `data()` to see the available data sets.

1. For `Theoph`, show the first few rows with the column order reversed.
2. For `Theoph`, change 'conc' to 'concentration'.
3. For `Theoph`, convert `Wt` to an integer.
4. For `ToothGrowth`, replace 'VC' and 'OJ' with 'vitamin C' and 'orange juice'.
5. For `Indometh`, convert mcg/mL to mg/mL in `conc`.
6. For `CO2`, what is the key?
7. For `ChickWeight`, what is the key?
8. For `Indometh`, what is the key?
9. For `Orange`, what is the key?
10. For `airquality`, what is the key?
11. For `beaver1`, what is the key?
12. For `trees`, what is the key?
13. Sort `Animals` on increasing body size.
14. For `Indometh`, impute all conc. values over 2 with the mean of the remaining values.
15. For `wtloss` (package MASS), impute all Weight after 200 days with the last observation.
16. For `women` (package datasets), calculate body mass index using `bmi()`.
17. For `Puromycin`, remove the 'state' column and the second row.
18. Merge 'Population' and 'Area' from `state.x77` with `USArrests`.
19. Merge `Animals` and `mammals` (both package MASS).
20. Add population density data from `road` to `state.x77`.

Chapter 7

Modeling in R

7.1 Objectives

After completing this chapter, you will be able to . . .

- Recognize and use the formula syntax that is common to most R modeling functions.
- Understand the role of factors with typical R modeling functions.
- Recognize and use some of the generic functions that can be used to interrogate typical model fit objects.
- Understand how missing values are handled by typical modeling functions in R.

7.2 Introduction

There are *many* model fitting functions in R. Without attempting to be at all comprehensive, here are some of the more commonly used modeling functions:

- `lm()` linear models (regression, ANOVA, ANCOVA) with fixed effects only and uncorrelated, Normally distributed, residuals
- `lme()` linear models with at least one random effect and Normally distributed residuals
- `gls()` general linear models, i.e. models with fixed effects only and Normally distributed, possibly correlated, residuals
- `nls()` nonlinear models with with fixed effects only and uncorrelated, Normally distributed, residuals

- `nlme()` nonlinear models with at least one random effect and Normally distributed residuals
- `glm()` general*ized* linear models ("linearizable" models with possibly non-Normal residuals, e.g. logistic regression)
- `survfit()` Cox regression (for time-to-event data)
- `gam()` generalized additive models (including, e.g. models with splines)

All of the above functions are available in either *base* or *recommended* R packages. The classic *Modern Applied Statistics with S and S-plus* by Venables and Ripley [3] provides an excellent survey of model fitting functions, including most of those mentioned above.

We will focus fairly specifically on functions that are consistently relevant in pharmacometric analyses, namely `lm`, `nls`, and `nlme`. This focus notwithstanding, we will see that there is a more or less generic framework for modeling in R. If you learn the basics of this framework, you will be well-positioned to teach yourself some of the other modeling functions that we will not have time to cover.

We are going to work with the same data set we used in our plotting chapters:

Listing 7.1:

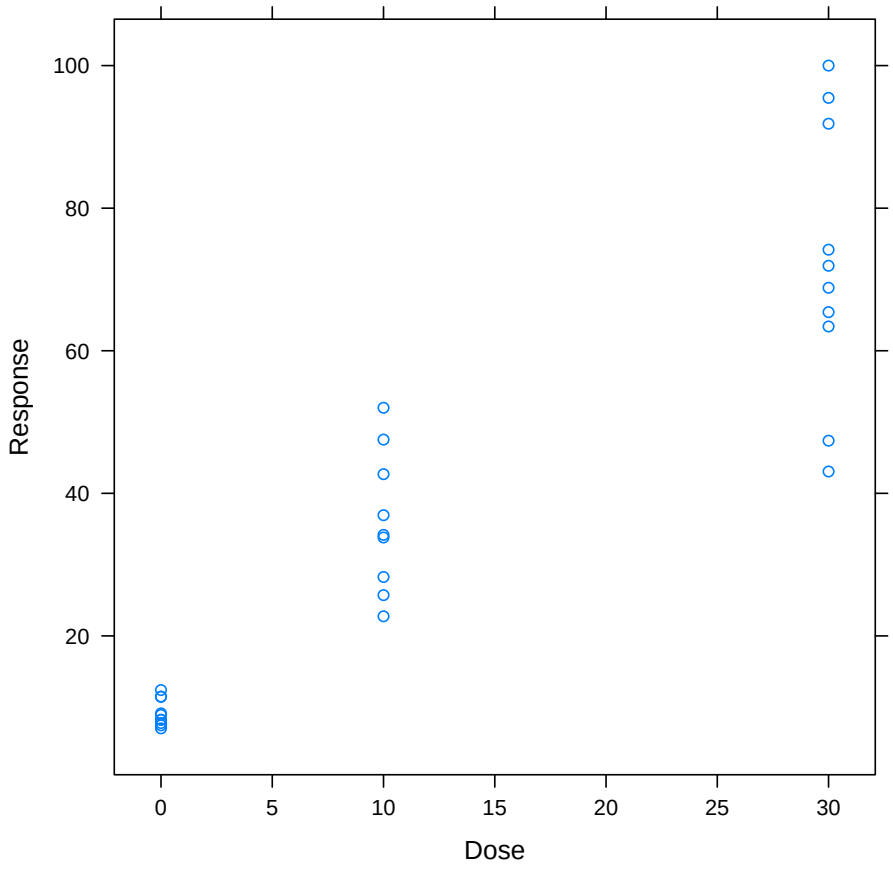
```
set.seed(1)
nSubj <- 40
doses <- c(0, 10, 30, 100)
times <- seq(0, 9, 3)
dat <- expand.grid(Subject = 1:nSubj,
                   Month = times)
dat <- as.data.frame(unclass(dat))
dat$Dose <- doses[as.numeric(dat$Subject) %% length(doses) + 1]
dat$Exposure <- dat$Dose * exp(rnorm(nrow(dat), 0, 0.3))
noise <- rep(rnorm(nSubj, 0, 0.2), length(doses)) + rnorm(nrow(dat
   ), 0, 0.2)
emax <- c(0, 50, 100, 100)[match(dat$Month, times)]
dat$Response <- with(dat, 10 + emax * Exposure / (Exposure + 25))
   * exp(noise)
dat$Response <- pmin(dat$Response, 100)
dat$Dose2 <- factor(paste(dat$Dose, "mg"))
dat$Dose2 <- factor(dat$Dose2, levels = levels(dat$Dose2)[c(1, 2,
   4, 3)])
dat$Sex <- rep(sample(c("M", "F"), nSubj, replace = TRUE), length(
   times))
dat$Response[dat$Month == 9 & dat$Subject == 1] <- NA
```

7.3 Fitting a regression model with lm()

We will start by focusing on a subset of the data, excluding the highest dose level and considering only observations from month 9:

Listing 7.2:

```
library(lattice)
print(
      xyplot(Response  ~  Dose,
             subset = Month == 9 & Dose < 40,
             data = dat)
      )
```



For this subset of data, a model describing `Response` as a linear function of `Dose` seems reasonable. That is, letting i be an index on patients,

$$Response_i = \alpha + \beta Dose_i + \varepsilon_i$$

From our plot, we can plainly see that the ε_i terms have different variances at the different dose levels, which violates the assumptions required for standard linear regression, but let's ignore this fact for now. We can fit a standard linear regression with:

Listing 7.3:

```
mod1 <- lm(Response ~ Dose,
           subset = Month == 9 & Dose < 40,
           data = dat)
```

Note the similarity of the `lm()` interface to the `xyplot()` interface: the response variable is on the left hand side of the formula and the explanatory variable is on the left hand side, and if we specify a `data` argument, `lm()` will know to look in that data frame for variables referenced in the formula and the subset argument (and we will see that there are some additional similarities). Most common modeling functions adhere to similar conventions. While these conventions are very helpful, some caution is in order: they are only conventions. For example, one should not necessarily expect that a syntax that works for `lm()` will also work with `lme()`, although that will often be the case.

We now have an object of class "lm":

Listing 7.4:

```
class(mod1)
```

```
[1] "lm"
```

As is typical of most R model objects, printing the model object itself is breathtakingly uninformative:

Listing 7.5:

```
mod1
```

```
Call:
lm(formula = Response ~ Dose, data = dat, subset = Month == 9 &
    Dose < 40)

Coefficients:
(Intercept)         Dose
     11.534        2.059
```

We will see in the next section how to get a more informative description of our model fit. For now, we can at least see/guess that a model has been fit with an intercept and a slope with respect to `Dose`. Note that we didn't explicitly request an intercept, but it was assumed

by default that we wanted one. To override the default and fit a model with no intercept, we include a `-1` term in the formula:

Listing 7.6:

```
lm(Response ~ -1 + Dose,
   subset = Month == 9 & Dose < 40,
   data = dat)
```

```
Call:
lm(formula = Response ~ -1 + Dose, data = dat, subset = Month ==
    9 & Dose < 40)

Coefficients:
 Dose
2.514
```

Similarly, if we wanted to fit a model with *only* an intercept term (not advisable for this data set!), we would do:

Listing 7.7:

```
lm(Response ~ 1,
   subset = Month == 9 & Dose < 40,
   data = dat)
```

```
Call:
lm(formula = Response ~ 1, data = dat, subset = Month == 9 &
    Dose < 40)

Coefficients:
(Intercept)
      39.23
```

7.4 Generic methods for model objects

It seems a bit intimidating to extract information from `mod1` "by hand" (try `str(mod1)` to see how complicated this object is), and indeed we don't have to. The information that we would typically want can be more easily and reliably extracted using some functions we now demonstrate. These functions are all referred to as "generic methods", because they can also be used for other types of model objects.

The generic method `summary()` gives us, among other things, *T*-tests ("Wald tests") for each coefficient in the model:

Listing 7.8:

```
summary(mod1)
```

```
Call:
lm(formula = Response ~ Dose, data = dat, subset = Month == 9 &
    Dose < 40)

Residuals:
    Min      1Q  Median      3Q     Max
-30.245  -4.475  -2.398   2.047  26.687

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  11.5338     3.4258   3.367  0.00230 **
Dose          2.0593     0.1854  11.106 1.43e-11 ***
---
Signif. codes:  0 âĂŸ***âĂŹ 0.001 âĂŸ**âĂŹ 0.01 âĂŸ*âĂŹ 0.05 âĂŸ.
   âĂŹ 0.1 âĂŸ âĂŹ 1

Residual standard error: 12.65 on 27 degrees of freedom
  (1 observation deleted due to missingness)
Multiple R-squared: 0.8204,     Adjusted R-squared: 0.8138
F-statistic: 123.4 on 1 and 27 DF,  p-value: 1.427e-11
```

If we want the traditional ANOVA *F*-tests, we can get this with the `anova` function:

Listing 7.9:

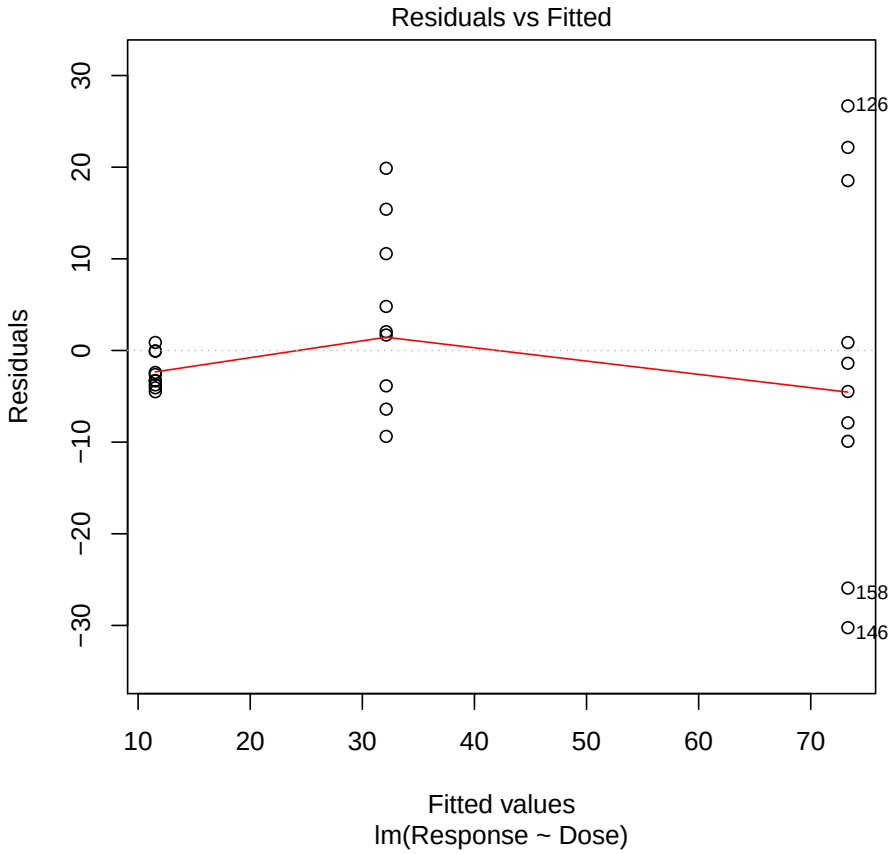
```
anova(mod1)
```

```
Analysis of Variance Table

Response: Response
          Df Sum Sq Mean Sq F value    Pr(>F)
Dose       1  19741   19741  123.35 1.427e-11 ***
Residuals 27   4321     160
---
Signif. codes:  0 âĂŸ***âĂŹ 0.001 âĂŸ**âĂŹ 0.01 âĂŸ*âĂŹ 0.05 âĂŸ.
   âĂŹ 0.1 âĂŸ âĂŹ 1
```

We get some decent diagnostic plots using the `plot()` function:

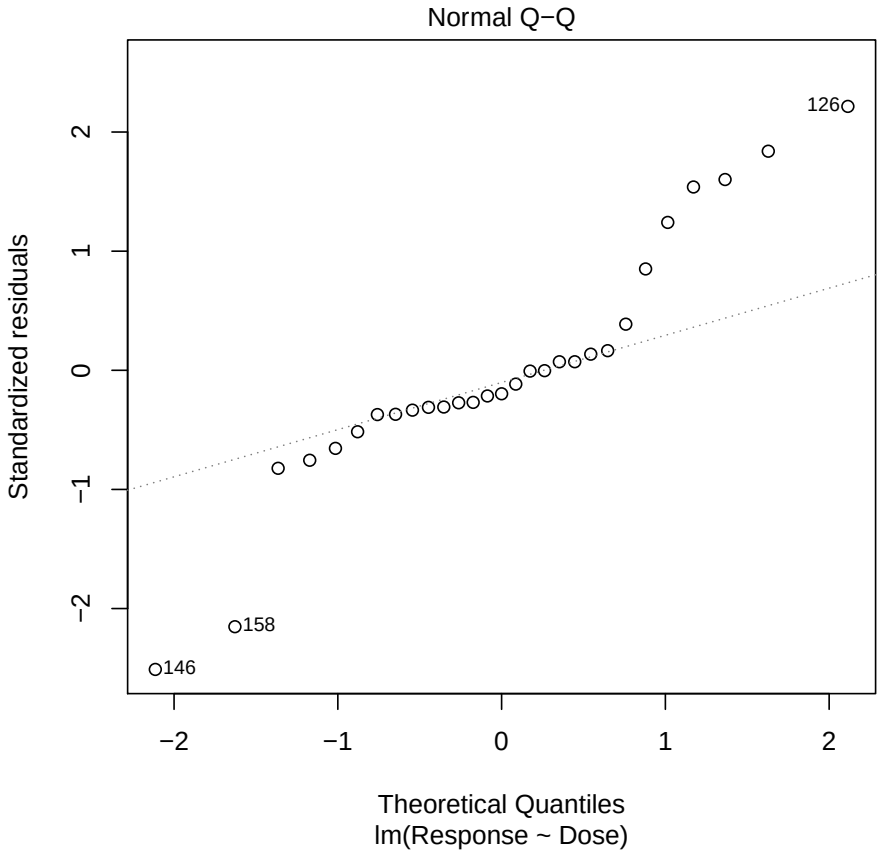
Listing 7.10:

```
plot(mod1, which = 1) # If using R interactively, try this without
    "which = 1"
```



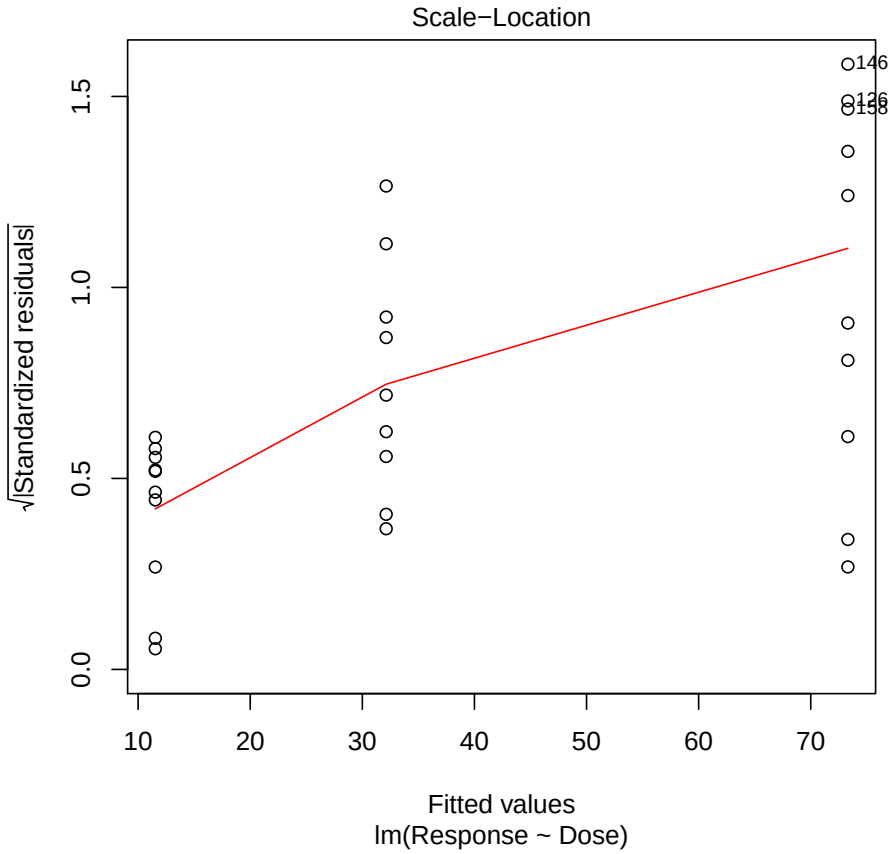
Listing 7.11:

```
plot(mod1, which = 2)
```



Listing 7.12:

```
plot(mod1, which = 3)
```



(In case we had not noticed it already, this last plot in particular calls attention to the unequal variances at the different dose levels.)

Additionally, there are generic methods to get estimated coefficients, confidence intervals, predicted values, and residuals, and model fit statistics such as AIC, among other things:

Listing 7.13:

```
coef(mod1)
```

```
(Intercept)        Dose
  11.533841    2.059294
```

Listing 7.14:

```
confint(mod1)
```

```
                2.5 %    97.5 %
(Intercept) 4.504658 18.563024
Dose        1.678854  2.439734
```

Listing 7.15:

```
predict(mod1)
```

```
     122      124      125      126      128      129
73.31267 11.53384 32.12678 73.31267 11.53384 32.12678
     130      132      133      134      136      137
73.31267 11.53384 32.12678 73.31267 11.53384 32.12678
     138      140      141      142      144      145
73.31267 11.53384 32.12678 73.31267 11.53384 32.12678
     146      148      149      150      152      153
73.31267 11.53384 32.12678 73.31267 11.53384 32.12678
     154      156      157      158      160
73.31267 11.53384 32.12678 73.31267 11.53384
```

Note: although we won't demonstrate it right here, `predict` can also be used to compute prediction intervals and confidence intervals for expected values.

Listing 7.16:

```
resid(mod1)
```

```
         122          124          125          126
 22.15691319  -3.31925088   4.80758926  26.68733372
         128          129          130          132
 -3.75206922  -3.85748658   0.86758696  -4.07105264
         133          134          136          137
 10.56422267  -4.47541738  -0.03564804  19.88075107
         138          140          141          142
 18.53727433  -0.08111633  -6.40211823  -1.39444908
         144          145          146          148
 -4.49646620  -9.37160127 -30.24470275  -3.27641519
         149          150          152          153
  1.68398482  -9.90695115   0.87539633   2.04677343
         154          156          157          158
 -7.89066912  -2.39772798  15.41324865 -25.92537333
         160
 -2.62255908
```

Listing 7.17:

```
AIC(mod1)
```

```
[1] 233.4132
```

Note that, by default, the vectors returned by `fitted()` and `resid()` do not include elements for the observation with the missing value. If we want to keep our fitted values and residuals aligned with our original data set, we need to tell `lm()` how to handle missing values. If we specify `na.action = na.exclude` (whereas the default is

`na.action = na.omit`), then `predict` and `resid` will pad their return vectors with `NA`s in the right places.

Listing 7.18:

```
mod1.1 <- lm(Response ~ Dose,
             subset =  Month == 9 & Dose < 40,
             data = dat,
             na.action = na.exclude)
```

Listing 7.19:

```
predict(mod1.1)
```

```
     121      122      124      125      126      128
      NA 73.31267 11.53384 32.12678 73.31267 11.53384
     129      130      132      133      134      136
32.12678 73.31267 11.53384 32.12678 73.31267 11.53384
     137      138      140      141      142      144
32.12678 73.31267 11.53384 32.12678 73.31267 11.53384
     145      146      148      149      150      152
32.12678 73.31267 11.53384 32.12678 73.31267 11.53384
     153      154      156      157      158      160
32.12678 73.31267 11.53384 32.12678 73.31267 11.53384
```

If we want to generate a non-missing prediction for our missing observation, we need to pretend that our original data set is "new data":

Listing 7.20:

```
predict(mod1.1,
        newdata = subset(dat,
          Month == 9 & Dose < 40,
          select = Dose)
        )
```

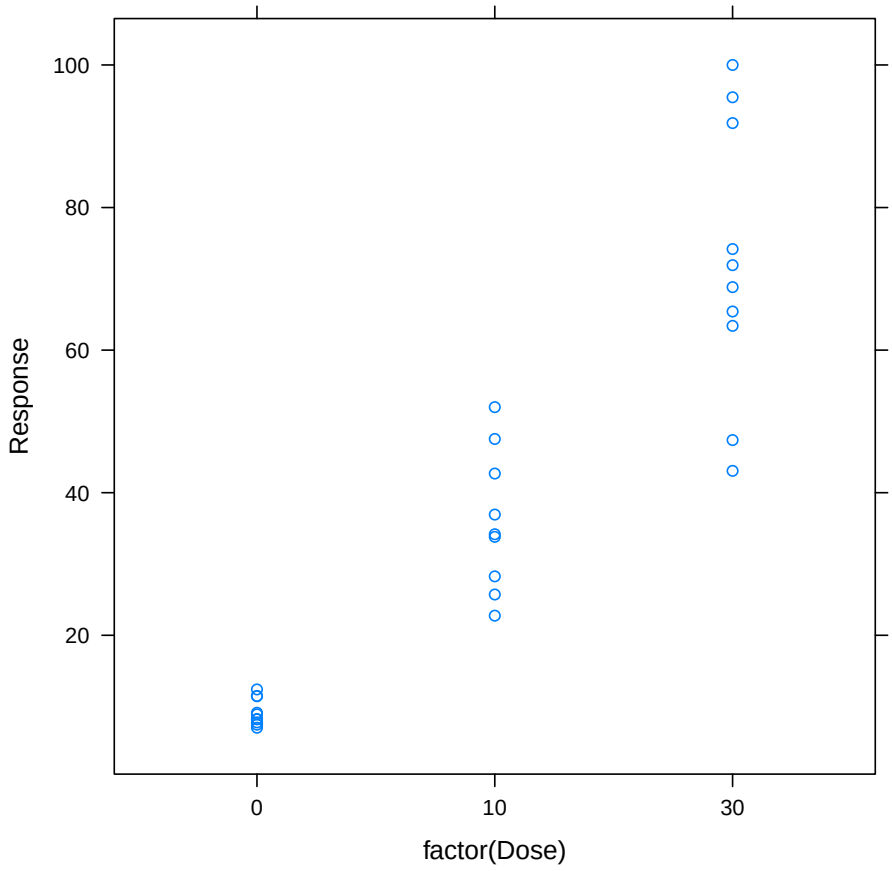
```
     121      122      124      125      126      128
32.12678 73.31267 11.53384 32.12678 73.31267 11.53384
     129      130      132      133      134      136
32.12678 73.31267 11.53384 32.12678 73.31267 11.53384
     137      138      140      141      142      144
32.12678 73.31267 11.53384 32.12678 73.31267 11.53384
     145      146      148      149      150      152
32.12678 73.31267 11.53384 32.12678 73.31267 11.53384
     153      154      156      157      158      160
32.12678 73.31267 11.53384 32.12678 73.31267 11.53384
```

7.5 Fitting an "ANOVA model" with lm()

Recall that the behavior of `xyplot()` depends on the *mode* of the variables we use:

Listing 7.21:

```
print(
      xyplot(Response ~ factor(Dose),
             subset = Month == 9 & Dose < 40,
             data = dat)
      )
```



This is also the case with `lm()`. If we give `lm()` a factor version of `Dose` (rather than a numeric version, as in the previous example), it will fit an ANOVA-type model, i.e.

$$Response_i = \mu + \beta_1 \mathrm{I}_{[Dose_i=10]} + \beta_2 \mathrm{I}_{[Dose_i=30]} + \varepsilon_i$$

(Again, the heterogeneity of variances violates the assumptions required for standard ANOVA, but for now we blissfully ignore this.)

Listing 7.22:

```
mod2 <- lm(Response ~ factor(Dose),
           subset = Month == 9 & Dose < 40,
           data = dat)
summary(mod2)
```

```
Call:
lm(formula = Response ~ factor(Dose), data = dat, subset = Month ==
    9 & Dose < 40)

Residuals:
     Min       1Q   Median       3Q      Max
-29.0859  -3.3166  -0.9587   2.2820  27.8462

Coefficients:
               Estimate Std. Error t value Pr(>|t|)
(Intercept)       9.216      3.981   2.315   0.0287 *
factor(Dose)10   26.773      5.784   4.629 8.94e-05 ***
factor(Dose)30   62.938      5.629  11.180 1.98e-11 ***
---
Signif. codes:  0 âĂŸ***âĂŹ 0.001 âĂŸ**âĂŹ 0.01 âĂŸ*âĂŹ 0.05 âĂŸ.
   âĂŹ 0.1 âĂŸ âĂŹ 1

Residual standard error: 12.59 on 26 degrees of freedom
  (1 observation deleted due to missingness)
Multiple R-squared: 0.8288,     Adjusted R-squared: 0.8156
F-statistic: 62.93 on 2 and 26 DF,  p-value: 1.086e-10
```

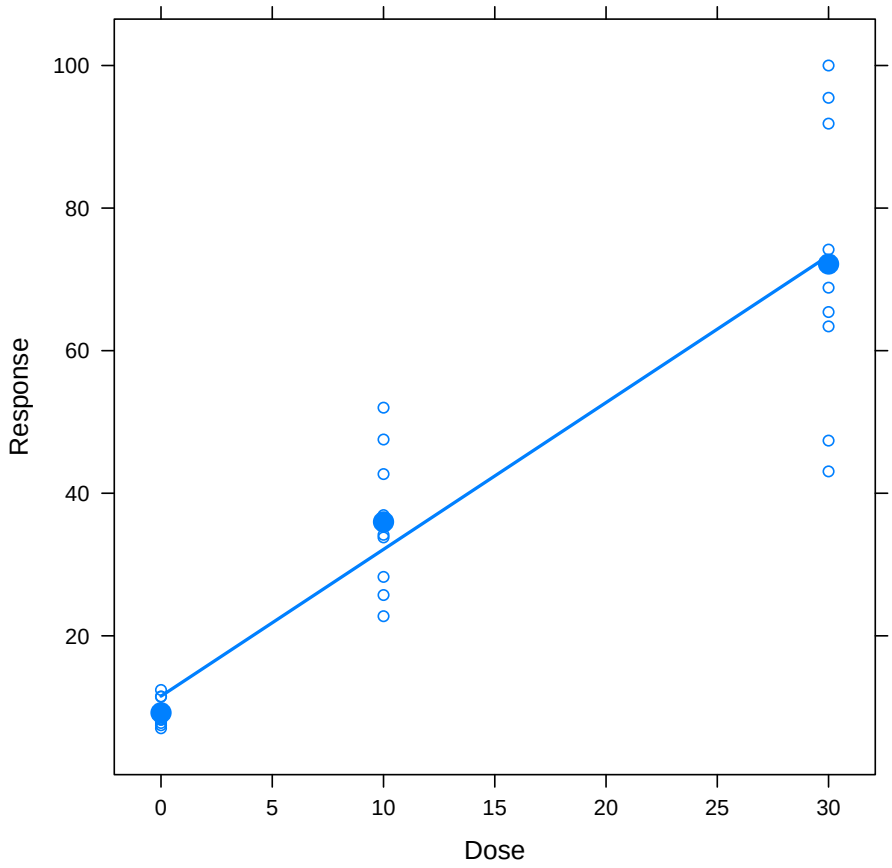
(As you may know from courses you have taken in regression, an ANOVA model is just a particular type of regression model using indicator variables. If for some reason you want to see those indicator variables, do: `model.matrix(mod2)`; compare this to what you get with `model.matrix(mod1)`.)

Let's make a plot showing the underlying data and the `mod1` and `mod2` fits:

Listing 7.23:

```
print(
      xyplot(Response ~ Dose,
             subset = Month == 9 & Dose < 40,
             data = dat,
             panel = function(x, y, ...) {
               panel.xyplot(x, y, ...)
```

```
                ux <- sort(unique(x))
                m1pred <- predict(mod1, newdata = data.frame(Dose =
                    ux))
                m2pred <- predict(mod2, newdata = data.frame(Dose =
                    ux))
                panel.xyplot(ux, m1pred, type = 'l', lwd = 2)
                panel.xyplot(ux, m2pred, pch = 19, cex = 1.5)
                }
            )
    )
```



As one can guess from looking at the output, `lm()` has used zero as the reference dose, and the model coefficients represent contrasts of the other two doses to the zero dose. This resulted from two distinct but related defaults. The first default that came into play was when zero became the first level of `factor(Dose)`:

Listing 7.24:

```
levels(factor(dat$Dose))[1]
```

```
[1] "0"
```

The second default that came into play was the use of “treatment contrasts". Meaningful coverage of contrasts would be somewhat technical and is beyond the scope of course. If you are interested in learning more, the help file `?contrasts` is a good place to start. For our purposes, the key take-home message is that models will be parameterized in terms of differences in means between factor levels, with the first factor level used as the reference level.

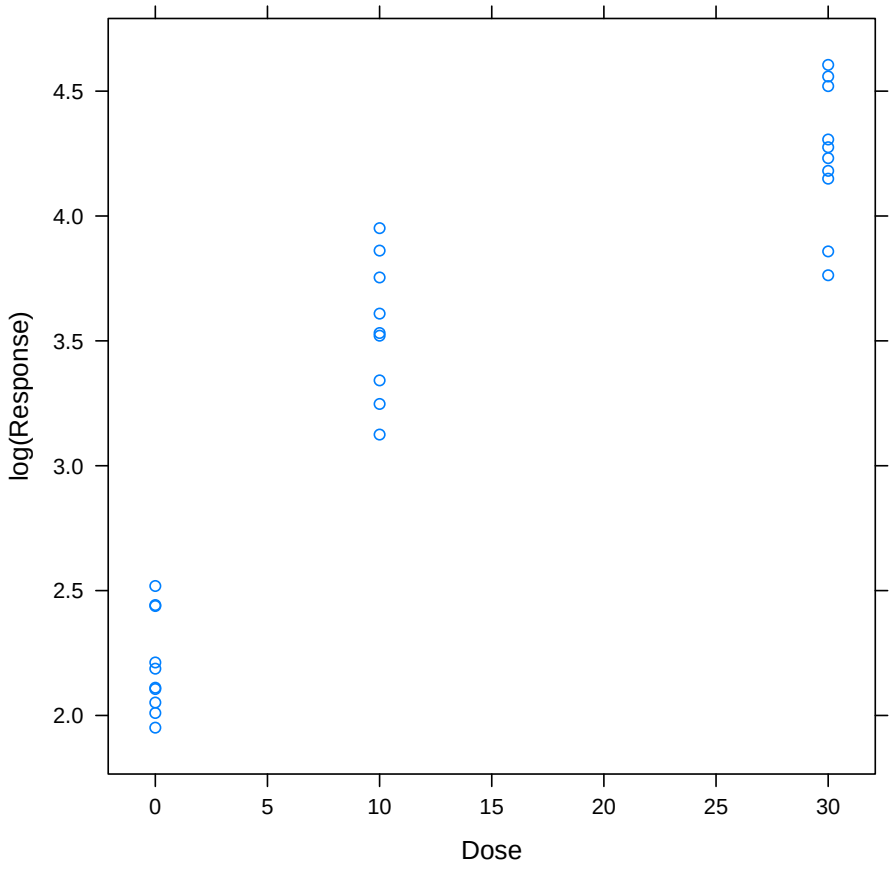
Interaction contrasts (e.g. if gender is one factor and age is another, and you want to compare elderly men to elderly women) can be particularly painful to work with using just functions from *base* and *recommended* packages. A very helpful and well-written package if you need to do this sort of thing is the “contrast” package (available on CRAN).

7.6 Fitting a nonlinear model with nls()

Our residuals versus predicted plot clearly showed that residuals from our regression model were not identically distributed. In this case a natural approach is to see if variances stabilize when we log-transform the response:

Listing 7.25:

```
print(
      xyplot(log(Response)  ~  Dose,
             subset =  Month == 9 & Dose < 40,
             data = dat
             )
      )
```



This looks pretty good in terms of variance stabilization, but now our mean values don't fall in a line. We could try to stay in the linear model world by applying a known transformation (e.g. log or square root) to `Dose`, but for illustration purposes, let's instead try a nonlinear model. A power relationship might be appropriate, i.e.:

$$\log(Response_i) = \alpha + Dose_i^{\beta} + \varepsilon_i$$

We can fit this with `nls()`. While `nls()` is now in the base *stats* package (which is loaded by default), some of the related methods (e.g. the method for plotting an `nls` object) are still in the *nlme* pakcage (which is not loaded by default). Therefore, my recommendation is to always load the *nlme* package whenever you are going to use `nls()`.

The syntax is similar to `lm()`, but we need to provide starting values for the iterative fitting algorithm. (There are also such things as "self-starting" models for `nls()`, for which one doesn't need to supply initial values, we will use these in an example in the section on `nlme()`; also, if you are interested, look at the functions in package *stats* whose names begin with "SS", e.g. SSlogis, SSmicmen, etc.)

Listing 7.26:

```
library(nlme)
mod3 <- nls(log(Response)  ~  alpha + Dose^beta,
            subset =  Month == 9 & Dose < 40,
            data = dat,
            start = list(alpha = log(10), beta = 0.5)
            )
```

We can now apply some of the same generic methods that we applied to our `lm` object:

Listing 7.27:

```
summary(mod3)
```

```
Formula: log(Response) ~ alpha + Dose^beta

Parameters:
      Estimate Std. Error t value Pr(>|t|)
alpha  2.12418    0.08309   25.56  < 2e-16 ***
beta   0.20802    0.01826   11.39 8.07e-12 ***
---
Signif. codes:  0 âĂŸ***âĂŹ 0.001 âĂŸ**âĂŹ 0.01 âĂŸ*âĂŹ 0.05 âĂŸ.
   âĂŹ 0.1 âĂŸ âĂŹ 1

Residual standard error: 0.282 on 27 degrees of freedom

Number of iterations to convergence: 6
Achieved convergence tolerance: 3.581e-07
  (1 observation deleted due to missingness)
```

Listing 7.28:

```
confint(mod3)
```

```
           2.5%     97.5%
alpha 1.9527145 2.2968216
beta  0.1672837 0.2437575
```

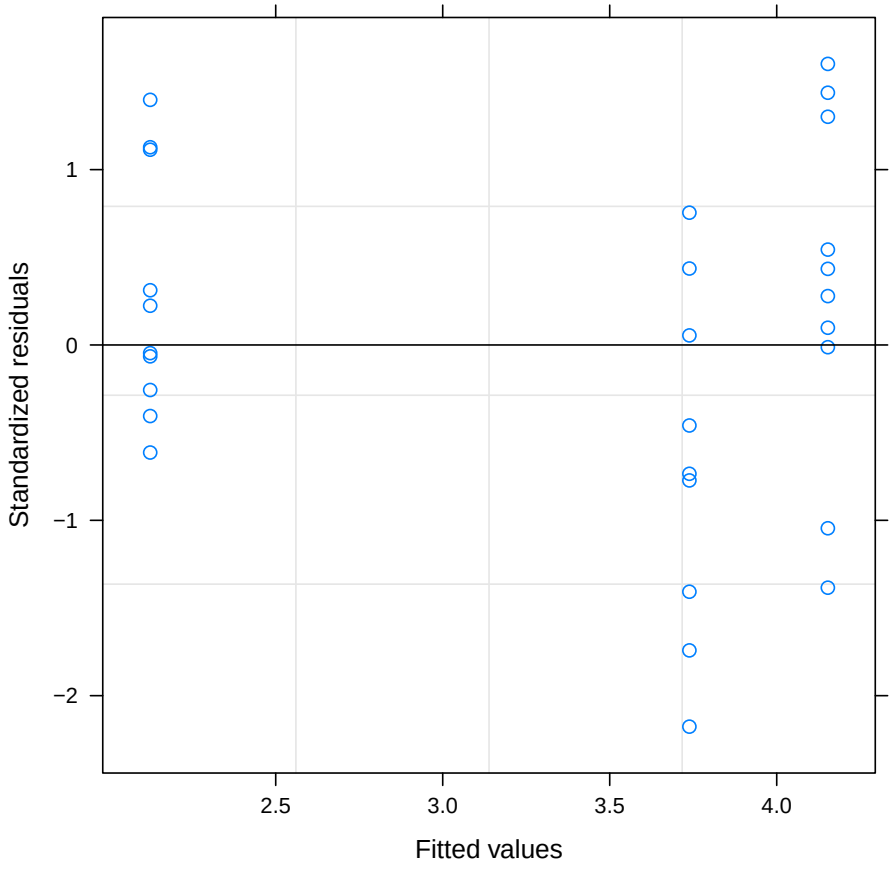
Listing 7.29:

```
AIC(mod3)
```

```
[1] 12.80844
```

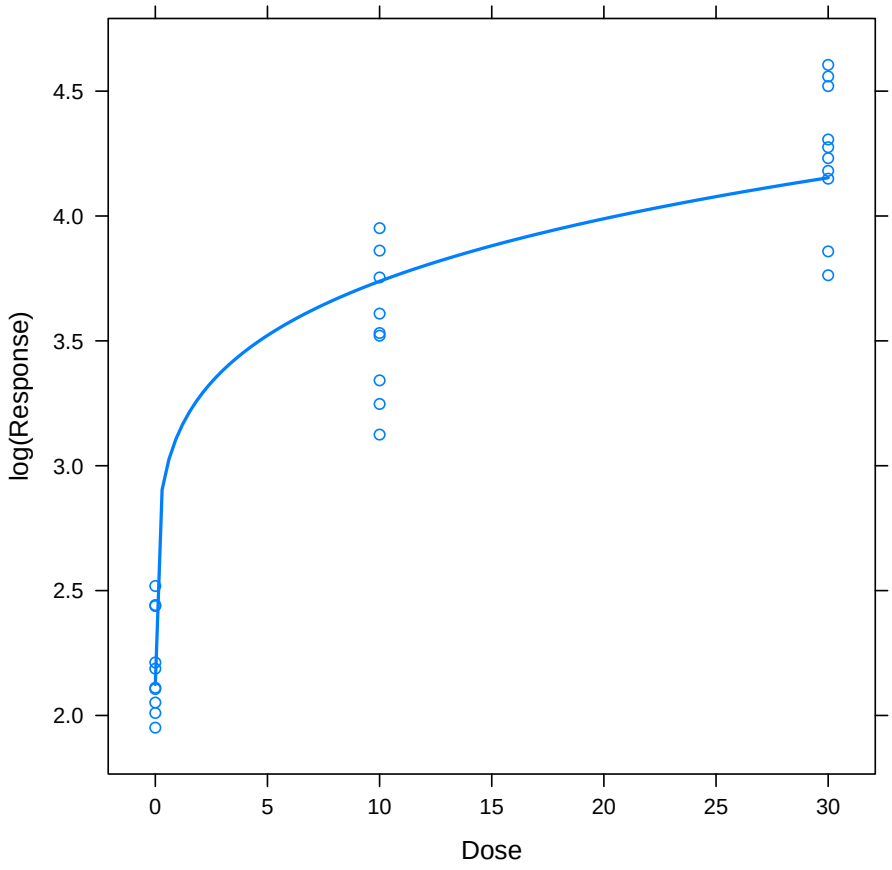
Listing 7.30:

```
print(
      plot(mod3)
      )
```



Listing 7.31:

```
print(
      xyplot(log(Response)  ~  Dose,
             subset =  Month == 9 & Dose < 40,
             data = dat,
             panel = function(x, y, ...) {
               panel.xyplot(x, y, ...)
               xx <- seq(min(x, na.rm = TRUE), max(x, na.rm = TRUE
                  ), length = 100)
               m3pred <- predict(mod3, newdata = data.frame(Dose =
                   xx))
               panel.xyplot(xx, m3pred, type = 'l', lwd = 2)
             }
             )
      )
```



We could certainly find a better fitting model for these data, but that's not the point of this lesson, so let's move on . . .

7.7 Fitting linear mixed-effects models with lme()

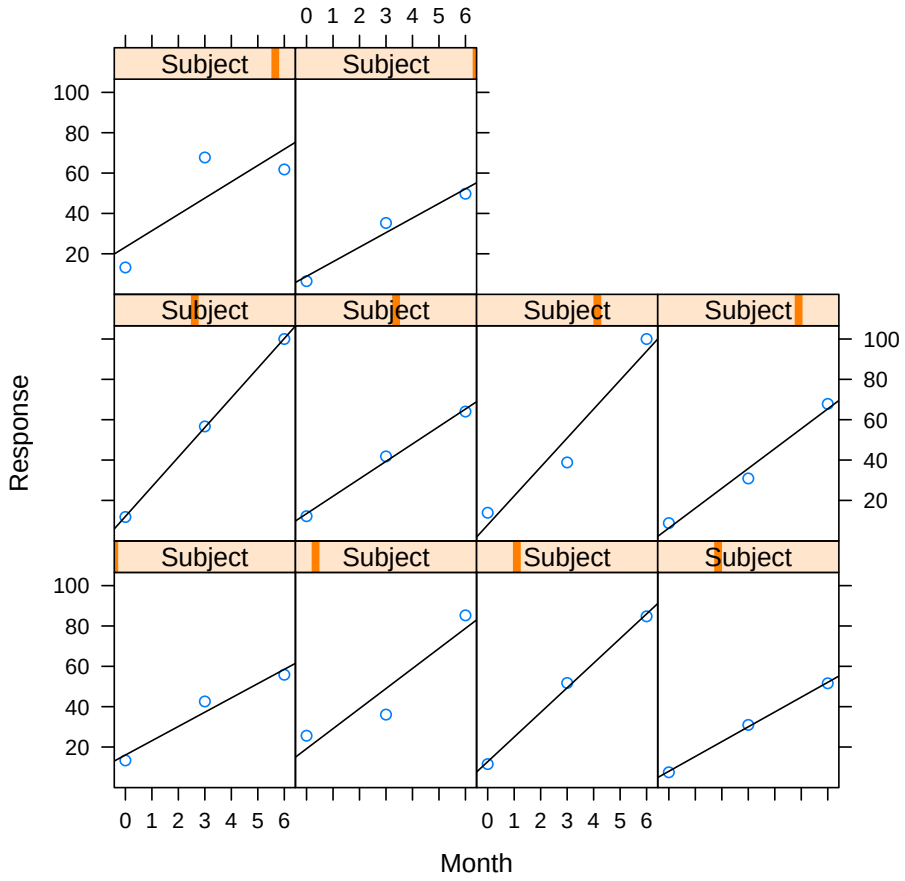
The progression of the response (on the original scale) appears to be pretty linear over the first six months. Suppose we wanted to estimate the rate of change of the response over the zero to six month time period for the 30 mg dose group.

First, let's plot the data with a trellis plot that conditions on subject:

Listing 7.32:

```
print(
      xyplot(Response ~ Month | Subject, data = dat,
             subset = Month < 9 & Dose == 30,
             panel = function(x, y, ...) {
```

```
                panel.xyplot(x, y, ...)
                panel.lmline(x, y, ...)
              }
              )
      )
```



Observations from the same individual at different time points are correlated, and our model should reflect this. One way to model this dependence is with subject-level random effects.

If we just want to add a random subject-level intercept, we could do the following:

Listing 7.33:

```
mod4 <- lme(fixed = Response ~ 1 + Month,
            data = dat,
            random = ~ 1 | Subject,
            subset = Month < 9 & Dose == 30
            )
```

```
mod4
```

```
Linear mixed-effects model fit by REML
  Data: dat
  Subset: Month < 9 & Dose == 30
  Log-restricted-likelihood: -115.4605
  Fixed: Response ~ 1 + Month
(Intercept)       Month
   12.76440     9.94345

Random effects:
 Formula: ~1 | Subject
        (Intercept) Residual
StdDev:    6.241444 11.59604

Number of Observations: 30
Number of Groups: 10
```

This model that we just fit is (letting j index time points, while i continues to index patients):

$$Response_{ij} = (\alpha + a_i) + \beta Month_i + \varepsilon_{ij}.$$

It would also be natural to consider random subject-level slopes, i.e.

$$Response_{ij} = (\alpha + a_i) + (\beta + b_i) Month_i + \varepsilon_{ij}.$$

This could be specified with:

Listing 7.34:

```
lme(Response ~ 1 + Month,
    data = dat,
    random = ~ 1 + Month | Subject,
    subset = Month < 9 & Dose == 30
    )
```

```
Error in lme.formula(Response ~ 1 + Month, data = dat, random = ~ 1
    + Month |  :
  nlminb problem, convergence error code = 1
  message = iteration limit reached without convergence (9)
```

As the error indicates, we have convergence problems with this model. At first blush this may be surprising: we have an admittedly small data set, but it does seem sufficient to support our

relatively simple model. In fact our model is not quite as simple as might be expected: our specification actually allows for arbitrary correlation between the random intercepts and the random slopes, that is:

$$\begin{pmatrix} a_i \\ b_i \end{pmatrix} \sim \text{Multivariate Normal}\left(\begin{pmatrix} 0 \\ 0 \end{pmatrix}, \begin{bmatrix} \sigma_a^2 & \rho\sigma_a\sigma_b \\ \rho\sigma_a\sigma_b & \sigma_b^2 \end{bmatrix}\right)$$

This is most likely an overparameterized model. A simpler model would require the random intercepts to be independent of the random slopes:

$$\begin{pmatrix} a_i \\ b_i \end{pmatrix} \sim \text{Multivariate Normal}\left(\begin{pmatrix} 0 \\ 0 \end{pmatrix}, \begin{bmatrix} \sigma_a^2 & 0 \\ 0 & \sigma_b^2 \end{bmatrix}\right)$$

Unfortunately, the specification of the random argument becomes a lot less intuitive in this case:

Listing 7.35:

```
lme(Response ~ 1 + Month,
            data = dat,
            random = list(Subject = pdDiag( ~ 1 + Month)),
            subset = Month < 9 & Dose == 30,
            na.action = na.exclude
            )
```

```
Linear mixed-effects model fit by REML
  Data: dat
  Subset: Month < 9 & Dose == 30
  Log-restricted-likelihood: -112.2853
  Fixed: Response ~ 1 + Month
(Intercept)       Month
   12.76440     9.94345

Random effects:
 Formula: ~1 + Month | Subject
 Structure: Diagonal
         (Intercept)    Month Residual
StdDev: 0.0006639948 2.513098 8.982265

Number of Observations: 30
Number of Groups: 10
```

In this case we have used `pdDiag` to indicate that we want the covariance matrix of the random effects to be a (p)ositive (d)efinite (Diag)onal matrix. Statistical theory beyond what

we have assumed for this course is prerequisite to really understanding this example in depth. Our goal for now is to make you aware of both the default behavior and flexibility of the `lme()` function. For a comprehensive treatment of these models and the *nlme* package, see Pinheiro and Bates [4].

While it is possible to get convergence with this last model, some further simplification is probably desirable (we are only working with three observations per subject over this time free, so we probably don't want to estimate two random effects for each subject). Noting that there is almost no between-subject variability in the intercept, it is probably best if we leave only the slope as random, forcing all subjects to share the same fixed intercept:

Listing 7.36:

```
mod5 <- lme(Response  ~  1 + Month,
            data = dat,
            random = list(Subject = pdDiag( ~  -1 + Month)),
            subset = Dose == 30,
            na.action = na.exclude
            )
mod5
```

```
Linear mixed-effects model fit by REML
  Data: dat
  Subset: Dose == 30
  Log-restricted-likelihood: -165.5926
  Fixed: Response ~ 1 + Month
(Intercept)       Month
  18.784719    6.933287

Random effects:
 Formula: ~-1 + Month | Subject
           Month Residual
StdDev: 1.227169 15.50113

Number of Observations: 40
Number of Groups: 10
```

Equivalently, we could specify this last model with:

Listing 7.37:

```
mod5 <- lme(Response  ~  1 + Month,
            data = dat,
            random =  ~  -1 + Month | Subject,
            subset = Dose == 30,
            na.action = na.exclude
            )
```

```
mod5
```

```
Linear mixed-effects model fit by REML
  Data: dat
  Subset: Dose == 30
  Log-restricted-likelihood: -165.5926
  Fixed: Response ~ 1 + Month
(Intercept)       Month
  18.784719    6.933287

Random effects:
 Formula: ~-1 + Month | Subject
           Month Residual
StdDev: 1.227169 15.50113

Number of Observations: 40
Number of Groups: 10
```

Prediction becomes a more subtle concept in the context of models with random effects. For our model there are two levels of predictions:

- "individual predictions", representing expected values of new observations in the *same* subjects under the same conditions (such new observations may not always be logistically or even logically possible, but the individual predictions usually will usually at least make sense in reference to a "thought experiment").
- "population predictions", representing expected values of new observations in *new* subjects from the same population, under the same conditions.

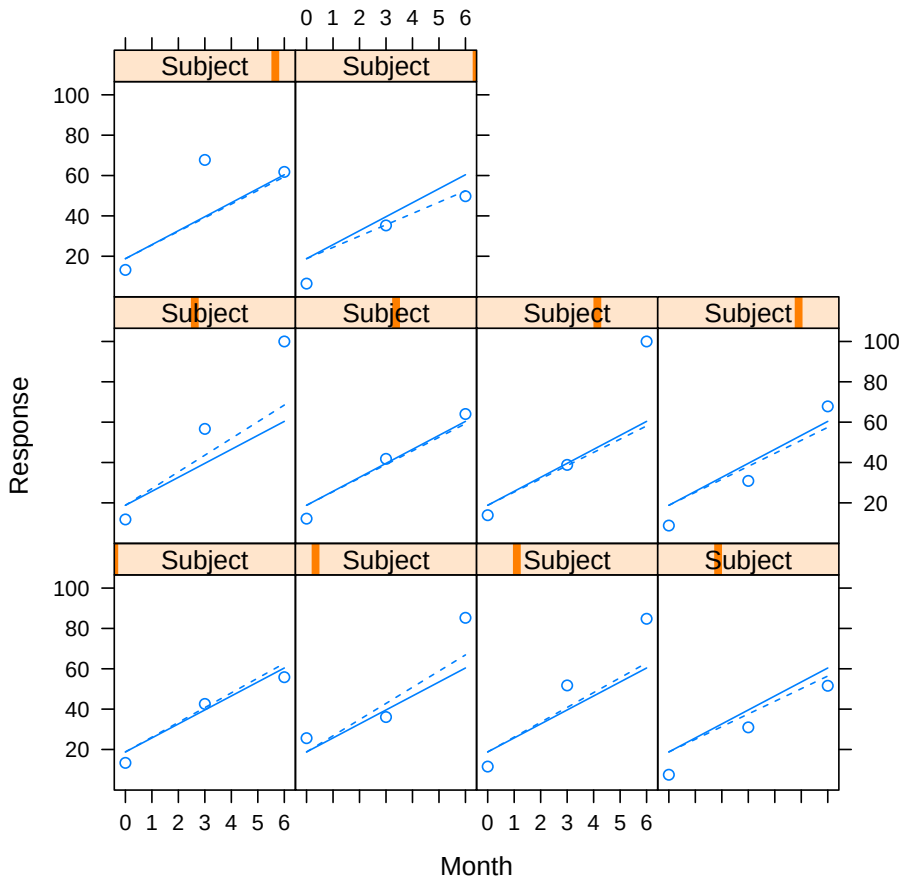
By default, the predict method for `lme()` will give us individual predictions. More generally for models with a hierarchy of random effects, the predict method will give us conditional predictions corresponding to the "innermost" level of the random effects. To get population predictions, we specify `level = 0` in our call to predict.

The following plot shows the difference between the two different types of predictions for this model:

Listing 7.38:

```
print(
      xyplot(Response ~ Month | Subject, data = dat,
             subset = Dose == 30 & Month < 9,
             panel = function(x, y, subscripts, ...) {
               panel.xyplot(x, y, ...)
               subjectData <- dat[subscripts, ]
               indPred <- predict(mod5, newdata = subjectData)
```

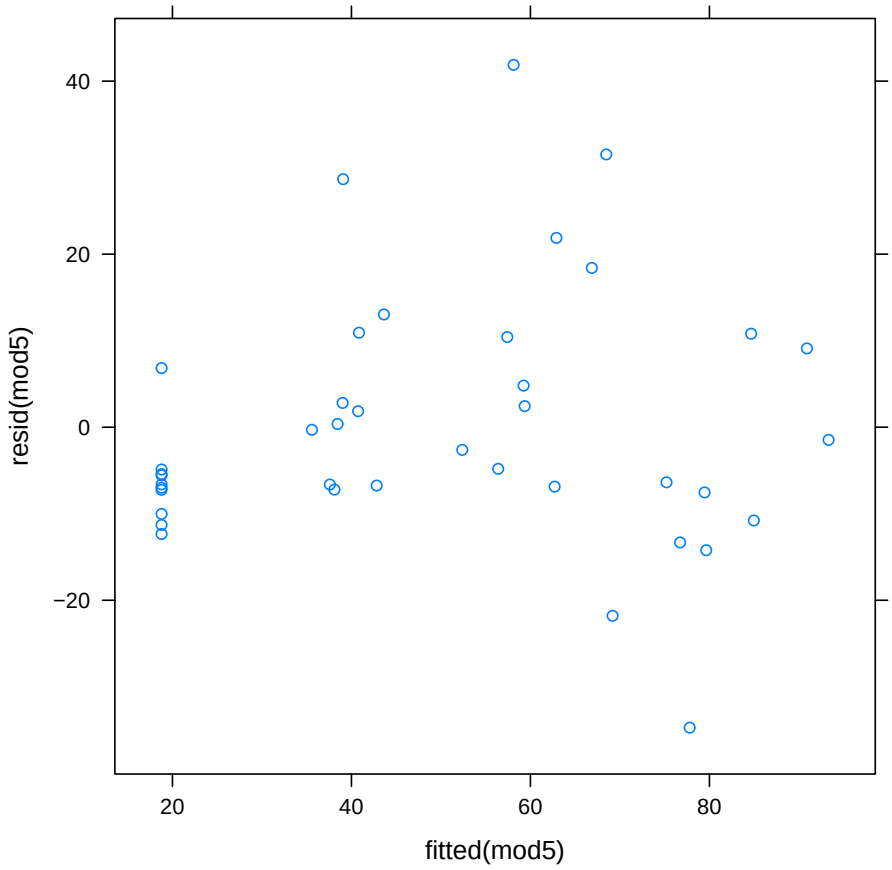
```
                popPred <- predict(mod5, newdata = subjectData,
                    level = 0)
                panel.xyplot(subjectData$Month, indPred, type = "l
                    ", lty = 2)
                panel.xyplot(subjectData$Month, popPred, type = "l
                    ", lty = 1)
            }
            )
    )
```



As with predictions, so with residuals:

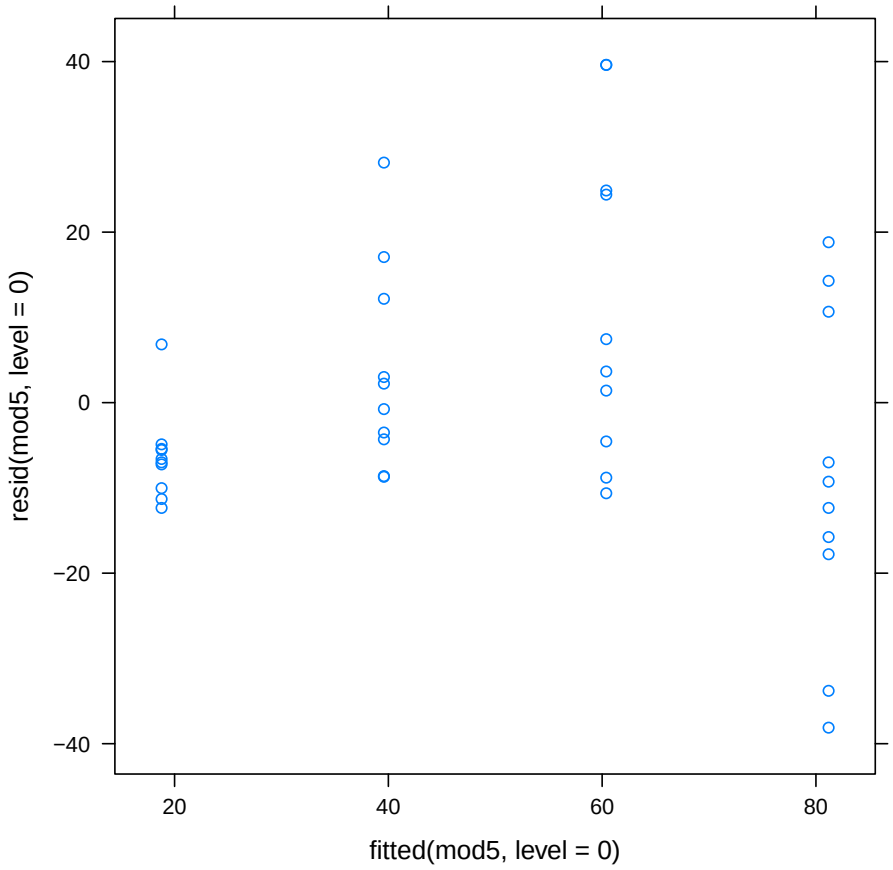
Listing 7.39:

```
print(
      xyplot(resid(mod5) ~ fitted(mod5))
      )
```



Listing 7.40:

```
print(
      xyplot(resid(mod5, level = 0) ~ fitted(mod5, level = 0))
      )
```



7.8 Fitting a nonlinear mixed effects model with nlme()

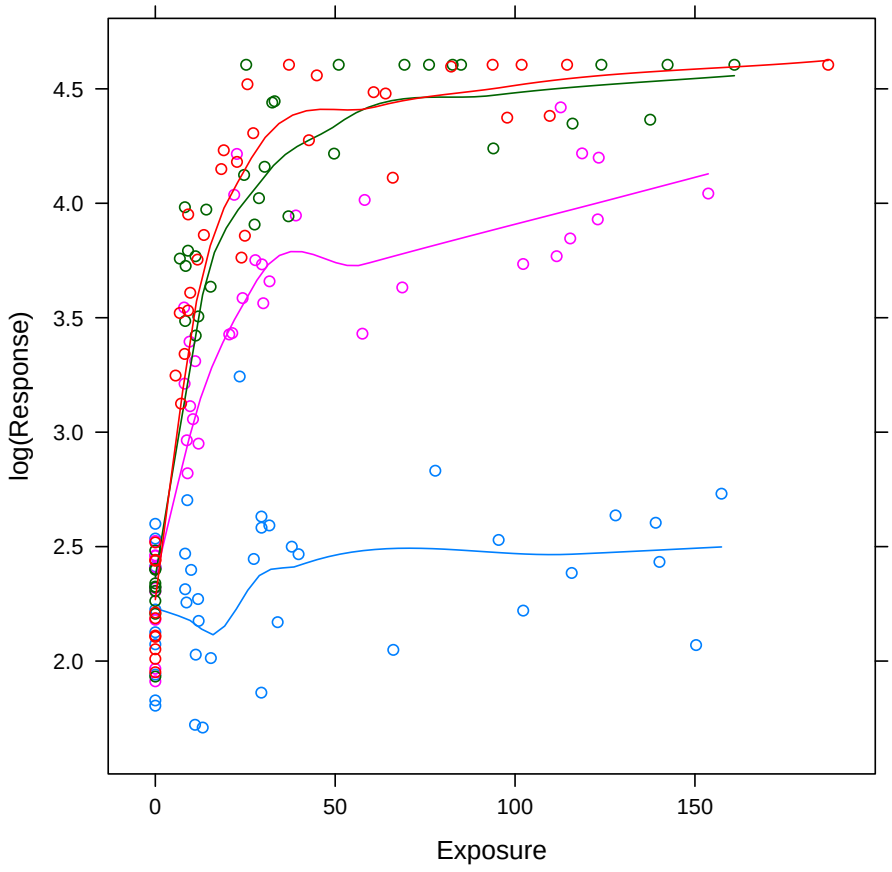
Exposition of this example will be very limited. As with the latter half of the last section, the goal here is primarily to demonstrate the flexibility of this class of models and its relevance to pharmacometric modeling. Those interested in learning more are again encouraged to spend some time with Pinheiro and Bates [4].

Let's remind ourselves what this data looks like when we plot it all together:

Listing 7.41:

```
print(
      xyplot(log(Response)  ~  Exposure,
             data = dat,
             groups = Month,
             panel = function(x, y, ...) {
               panel.superpose(x, y, ...)
```

```
            },
            panel.groups = function(x, y, ...) {
              panel.xyplot(x, y, ...)
              panel.loess(x, y, ...)
            }
            )
    )
```

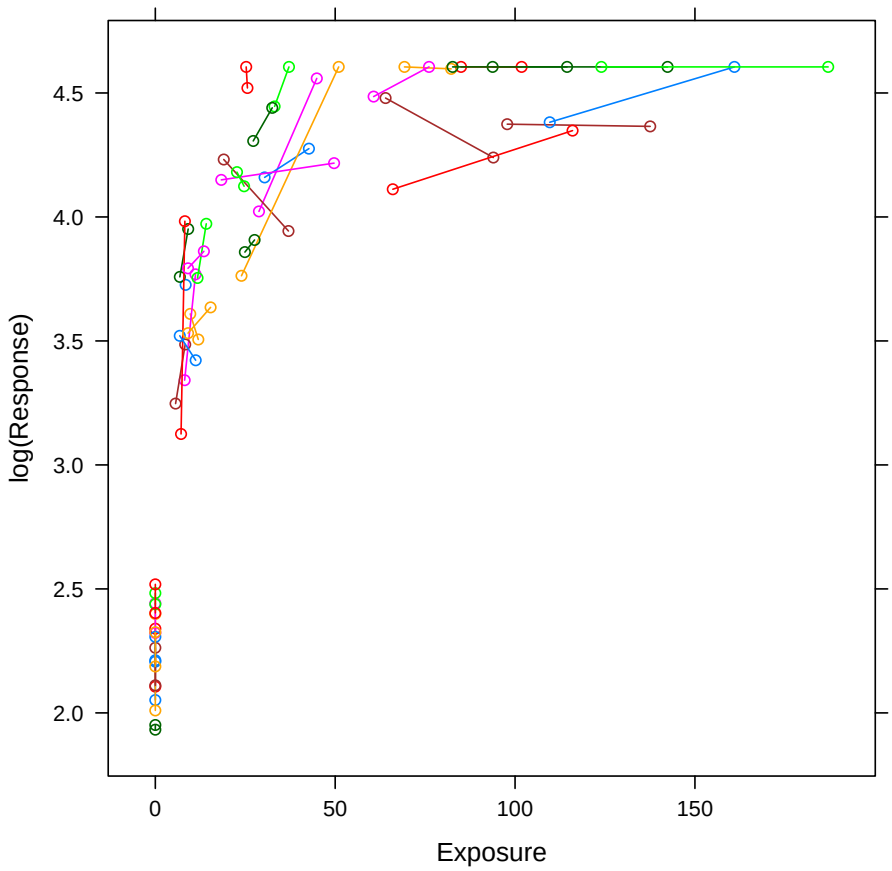


The loess fits suggest that, at least within each month, it woud be reasonable to describe the relationship between `Exposure` and `Response` using an asymptotic regression model (i.e. so that `Response` approaches some horizontal asymptote as `Exposure` goes to infinity).

Another consideration is that, as soon as we start modeling multiple months simultaneously, we are going to need to account for correlations amongst observations. For example, if we focus on months 6 and 9, we would have correlated pairs of observations, as suggested by the following plot:

Listing 7.42:

```
print(
      xyplot(log(Response)  ~  Exposure,
             data = dat,
             subset = Month %in% c(6, 9),
             groups = Subject,
             type = 'b'
             )
      )
```



The models fit by `nlme()` can be extremely complex, so it is sometimes a reasonable strategy to first ignore correlations to some approximate guesswork with `nls()` (corresponding to a "naïve pooling" approach), then move to `nlme()` only once you are comfortable that you are in the right ballpark. In this case we are going to start by using `nls()` to fit a model to the data from a single month, before moving to `nlme()` to model data over time.

We are going to use a "self-starting" version of an asymptotic regression model, encoded as

`SSasymp`. When using a self-starting model with `nls()`, it is not necessary to supply initial values:

Listing 7.43:

```
tmp1 <- nls(log(Response)  ~  SSasymp(Exposure, Asym, R0, lrc),
            data = dat,
            subset = Month %in% c(6, 9)
            )
tmp1
```

```
Nonlinear regression model
  model:  log(Response) ~ SSasymp(Exposure, Asym, R0, lrc)
   data:  dat
  Asym     R0    lrc
 4.448  2.267 -2.303
 residual sum-of-squares: 3.432

Number of iterations to convergence: 5
Achieved convergence tolerance: 5.611e-06
```

Let's see if this looks reasonable:

Listing 7.44:

```
print(
      xyplot(log(Response)  ~  Exposure,
             data = dat,
             subset = Month %in% c(6, 9),
             panel = function(x, y, ...) {
               panel.xyplot(x, y, ...)
               xo <- sort(x)
               yo <- y[order(x)]
               panel.xyplot(xo, predict(tmp1, newdata = data.frame
                  (Exposure = xo)), type = 'l')
             }
             )
      )
```

Now if we want to model this data over time, the model will need to change in two ways: we will want to model the dependence of observations within individuals, and we will want to allow the mean responses to vary with time. (For this fake data set, pharmacodynamic response lags months behind the PK; `Exposure` actually has no ability to explain the variation over time).

One way to dealing with the dependence issue is to introduce subject-level random effects on some of the parameters. Specifically, let's allow the intercept and the asymptote to vary by

subject (we will leave the other parameter fixed because we are not working with a lot of data here).

Here we add the six month data into the mix, but we don't yet want to add the other months, because our model doesn't yet allow the means to vary with time:

Listing 7.45:

```
mod6 <- nlme(model = log(Response) ~ SSasymp(Exposure, Asym, R0,
   lrc),
             data = dat,
             subset = Month %in% c(6, 9),
             fixed = list(Asym ~ 1, R0 ~ 1, lrc ~ 1),
             random = list(Asym ~ 1, R0 ~ 1),
             groups = ~ Subject,
             start = coef(tmp1),
             na.action = na.exclude
             )
mod6
```

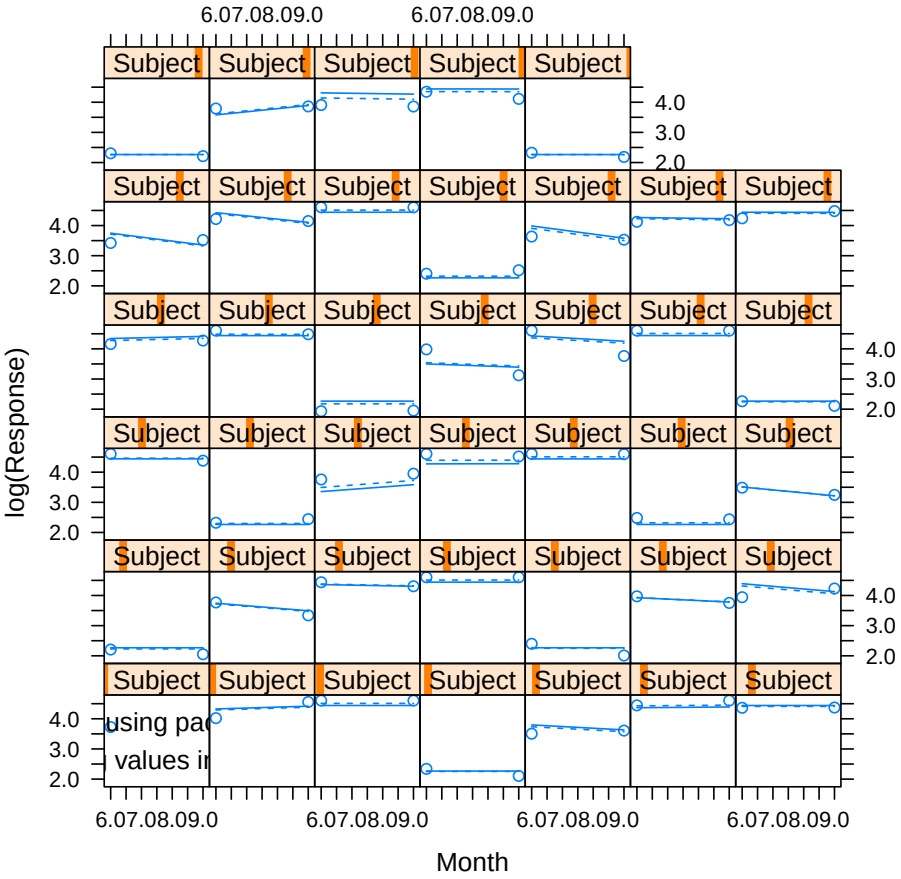
```
Nonlinear mixed-effects model fit by maximum likelihood
  Model: log(Response) ~ SSasymp(Exposure, Asym, R0, lrc)
  Data: dat
  Subset: Month %in% c(6, 9)
  Log-likelihood: 12.89803
  Fixed: list(Asym ~ 1, R0 ~ 1, lrc ~ 1)
     Asym        R0       lrc
 4.441109  2.265896 -2.283917

Random effects:
 Formula: list(Asym ~ 1, R0 ~ 1)
 Level: Subject
 Structure: General positive-definite, Log-Cholesky parametrization
         StdDev     Corr
Asym     0.11108420 Asym
R0       0.08006043 0.998
Residual 0.18268187

Number of Observations: 79
Number of Groups: 40
```

(For at least some usages of `nlme()`, we need to supply initial values even when using a "self-starting" model; "self-starting" is obviously a misnomer in this case.)

Listing 7.46:

```
print(
```

```
xyplot(log(Response) ~ Month | Subject, data = dat,
       subset = Month %in% c(6, 9),
       panel = function(x, y, subscripts, ...) {
         panel.xyplot(x, y, ...)
         subjectData <- dat[subscripts, ]
         indPred <- predict(mod6, newdata = subjectData)
         popPred <- predict(mod6, newdata = subjectData,
            level = 0)
         panel.xyplot(subjectData$Month, indPred, type = "l
            ", lty = 2)
         panel.xyplot(subjectData$Month, popPred, type = "l
            ", lty = 1)
       }
       )
)
```



Finally, let's allow the population mean of the intercept and asymptote to vary by Month,

so that we can throw all the data into the mix. (From the point of view of physiological interpretation, it would be unsatisfying to have this exposure-response asymptote vary as a function of month, but one can imagine that there might be a hidden physiological time-varying covariate for which month is a decent proxy.)

We do this by modifying the fixed component of this model:

Listing 7.47:

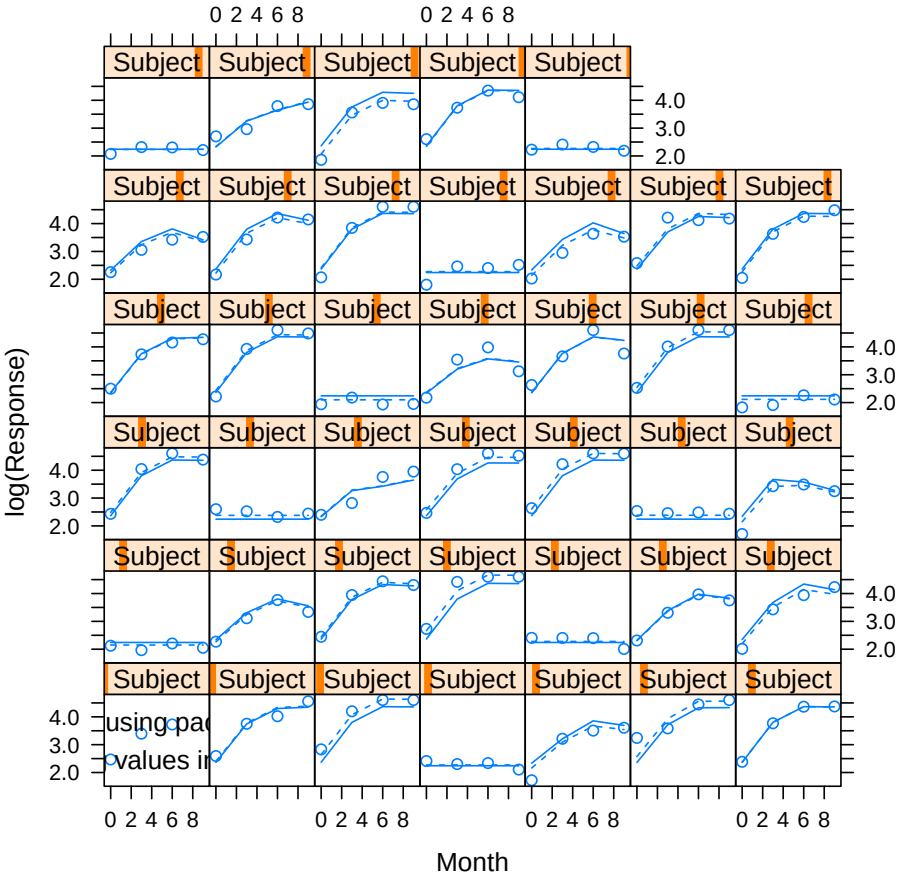
```
p <- coef(tmp1)
mod7 <- nlme(model = log(Response) ~ SSasymp(Exposure, Asym, R0,
   lrc),
             data = dat,
             fixed = list(Asym ~ factor(Month), R0 ~ 1, lrc ~
                1),
             random = list(Asym ~ 1, R0 ~ 1),
             groups = ~ Subject,
             start = c(p["Asym"], 0, 0, 0, p[c("R0", "lrc")]),
             na.action = na.exclude
             )
mod7
```

```
Nonlinear mixed-effects model fit by maximum likelihood
  Model: log(Response) ~ SSasymp(Exposure, Asym, R0, lrc)
  Data: dat
  Log-likelihood: 5.053913
  Fixed: list(Asym ~ factor(Month), R0 ~ 1, lrc ~ 1)
   Asym.(Intercept) Asym.factor(Month)3 Asym.factor(Month)6
          2.367357            1.433793            1.998596
Asym.factor(Month)9                  R0                 lrc
          1.989683            2.241898           -2.125852

Random effects:
 Formula: list(Asym ~ 1, R0 ~ 1)
 Level: Subject
 Structure: General positive-definite, Log-Cholesky parametrization
                 StdDev    Corr
Asym.(Intercept) 0.2092539 As.(I)
R0               0.1211240 -0.999
Residual         0.2008435

Number of Observations: 159
Number of Groups: 40
```

This appears to do something reasonable:

Listing 7.48:

```
print(
      xyplot(log(Response)  ~  Month | Subject, data = dat,
             panel = function(x, y, subscripts, ...) {
               panel.xyplot(x, y, ...)
               subjectData <- dat[subscripts, ]
               indPred <- predict(mod7, newdata = subjectData)
               popPred <- predict(mod7, newdata = subjectData,
                  level = 0)
               panel.xyplot(subjectData$Month, indPred, type = "l
                  ", lty = 2)
               panel.xyplot(subjectData$Month, popPred, type = "l
                  ", lty = 1)
             }
             )
      )
```



7.9 Homework

Use the month 9 data to fit an ANOVA model to the log transformed response. Do some basic model diagnostics and model summary, including characterizing treatment differences on the original scale.

Chapter 8

Modeling Outside of R

8.1 Objectives

After completing this chapter, you will be able to ...

- Understand the usage of the `metrumrg` package
- Understand the usage of `metaSub()` and `resample()`
- Read in and plot results of a population PK analysis from NONMEM
- Read in and plot results of a simple WinBUGS analysis
- Understand the usage of `rlog()`

8.2 Introduction

The varied nature of the data that are analyzed by the pharmacometrician necessitate the availability and use of various software applications. More often than not, each application has a set of unique data requirements that are not easily interchangeable. Each application also varies greatly as to the content of the output, ease of use of the output, and post-processing capabilities. The utility of R for preparing data has been demonstrated in the "Data Assembly" chapter and will be expanded upon in the upcoming chapter "Advanced Data manipulation". In this chapter we will explore some techniques for evaluating the ouput of some common applications (NONMEM, WinBUGS) utilized by the pharmacometrician. Instructions for developing models in NONMEM and WinBUGS is outside the scope of this chapter so instruction and examples will focus on pre- and post-processing steps in R. We will utilize a number of the functions available in the `metrumrg` package.

Listing 8.1:

```
library(metrumrg)
```

8.3 Usage of the metrumrg package

The `metrumrg` package was developed at Metrum Institute to provide a suite of tools for data set preparation, running NONMEM (locally or on a cluster), creating post-run plots and tables, creating logs for tracking a NONMEM analysis, and simulation and model based inference. Source code is available from GoogleCode (http://metrumrg.googlecode.com). We can access the list of available `metrumrg` functions from the RStudio help button. (The `metrumrg` package assumes that NONMEM has been installed using nmqual (http://nmqual.googlecode.com/ the open source NONMEM installation tool from Metrum Institute (http://metruminstitute.org).)

We will explore the `NONR72()` , `PLOTR()` , `rlog()`, `resample.data.frame`, and `metaSub.character()` functions. The workhorse of the metrumrg package is the `NONR72()` function. We can review the arguments for `NONR72()`.

Listing 8.2:

```
args(NONR72)
```

```
function (run, command, project = getwd(), boot = FALSE,
    grid = boot, concurrent = grid, urgent = !boot, udef = FALSE,
    invisible = udef, compile = TRUE, execute = TRUE,
    split = FALSE, checkrunno = TRUE, checksum = TRUE,
    diag = !boot, fdata = TRUE, logtrans = FALSE,
    nice = TRUE, epilog = NULL, dvname = NULL, grp = NULL,
    grpnames = NULL, cont.cov = NULL, cat.cov = NULL,
    par.list = NULL, eta.list = NULL, missing = -99, ...,
    interface = "autolog.pl", q = "all.q")
NULL
```

We will not go through all of the arguments but we will hit on a few relevant ones. If we wanted to execute a NONMEM run in our directory for this weeks lecture we would execute the code block below.

Listing 8.3:

```
NONR72(project = "/home/billk/MI205/modelingoutr/", command="/opt/
    NONMEM/nm72gf/nmqual/autolog.pl", run= c(1), grid=TRUE )
```

This function call and set of arguments represents the minimum information necessary for a successful `NONR72()` call. We basically need to tell `NONR72()` what directory contains the relevant control stream, the perl script being used to call NONMEM, and the number of

the control stream(s). At this point, the `NONR72()` function checks to ensure that the control stream (1.ctl) exists and exits with a message to the R console upon failure. If 1.ctl exists, NONMEM is executed using the specified control stream. Upon successful completion of NONMEM, the output files are used by `PLOTR()` to create a set of diagnostic plots.

Listing 8.4:

```
NONR72(project = "/home/billk/MI205/modelingoutr/", command="/opt/
   NONMEM/nm72gf/nmqual/autolog.pl", run= c(1), grid=TRUE )
```

As is demonstrated from the output of the above command on RStudio, NONMEM is installed and has run successfully. Assuming the 1.ctl file was setup correctly, we would have an output file and set of diagnostic plots that can be viewed to judge the adequacy of the model we fit to the data.

8.4 Preparing input files and data for modeling programs

For this chapter, the terminology "external modeling programs" will be used to refer to any software other than R. In many cases, the "controlling" parameters for external modeling programs reside in a simple ASCII text file that is read and interpreted during or immediately prior to execution. For example, the control stream in NONMEM and the model file in WinBUGS are simple ASCII text files. An entire analysis may result in 30+ NONMEM control streams not including the model evaluation steps (bootstrapping, predictive check) that may add 500 or 1000 more. For WinBUGS, the number of model files would likely be less. In either situation, the pharmacometrician is left with the unenviable task of editing and tracking changes to a large number of the "controlling" files. It would be preferable to edit the files programmatically so the process could be easily tracked and reproduced. This could be accomplished via an R script that utilizes external command-line programs like grep, sed, and awk to edit and save the relevant files. However, we would still be relying on programs outside of R that may or may not be available on all operating systems. Another approach would be to use the `metaSub()` in the `metrumrg` package. Lets take a look at the arguments and help file for the `metaSub()` function. (`metaSub()` is a generic function with methods available for character items within a file and/or entire files so we will ask for the arguments for the character class.)

Listing 8.5:

```
args(metaSub.character)
```

```
function (x, names, pattern = NULL, replacement = NULL,
    out = NULL, suffix = ".txt", fixed = TRUE, ...)
NULL
```

The key portions of this function are the information passed to the `pattern` and `replacement` arguments. These arguments can accept a simple character vector or a `list` of substitutions that may then contain a single or set of "regular" expressions to be evaluated. For this discussion we will limit these arguments to a character vector containing multiple pattern and replacement arguments. (The use of regular expressions will not be covered but an example piece of R code will be provided demonstrating their use in `metaSub`.)

Lets use `metaSub()` to generate additional control streams using the 1.ctl file as a template. We will want to take a quick look at 1.ctl to understand what we want to replace. For this example, 1.ctl will represent the final model and we are going to bootstrap the data set (more on this later) and use the final model against the new data set. We will want to replace the original data set name (found in $DATA) and remove the $TABLE items. (It is not important for todays exercise to understand why we are altering these items but it is important to understand how to do it.) Lets take a look at the portion of R code below.

Listing 8.6:

```
# metaSub() is part of the metrumrg package and will need to be
   loaded via the library command
# library(metrumrg)

metaSub(
        as.filename("1.ctl"),
        names=100,
        pattern=c("ex1_R.csv",
                          "$TABLE"),
        replacement=c("1.csv",
                      ";$TABLE"),
        out=".",
        suffix = ".ctl"
        )
```

In the "modelingoutr" directory there is now a new contol stream (100.ctl) that is an exact copy of 1.ctl with the exception of the items we changed in the R code above. A more realistic bootstrap would be to fit the model to 100 or even 1000 resampled data sets. The snippet of R code below will generate 10 new control streams but could easily be modified to generate 100 or even 1000 control streams.

Listing 8.7:

```
metaSub(
        as.filename("1.ctl"),
        names=100:110,
        pattern=c("ex1_R.csv",
                        "$TABLE"),
        replacement=list(expression(paste(as.numeric(name), ".csv
           ",sep=""")),
```

```
                          expression(paste(";$TABLE"))
                         ),
        out=".",
        suffix = ".ctl"
        )
```

The main difference in these 10 control streams is the name in the $DATA block. The name of the data set in each new control stream matches the numeric portion of the file name. for example, in 100.ctl the data set name is 100.csv. A `metaSub` call using regular expressions to perform a similar task in a slightly different way is provided as a reference example below.

Listing 8.8:

```
metaSub(
        as.filename("1.ctl"),
        names=100:110,
        pattern=list("\\$DATA[^$]*",
                          "\\$TABLE[^$]*"
                ),
        replacement=list(expression(paste("$DATA ../",as.numeric(
           name), ".csv \n",sep="")),
                                        expression(paste(";$TABLE")
                                           )
                         ),
        out=".",
        fixed=FALSE,
        suffix = ".ctl"
        )
```

The input file to `metaSub()` could be any ASCII file so its use is not limited to NONMEM control streams. The example below uses `metaSub` to revise a WinBUGS model file.

Listing 8.9:

```
metaSub(
        as.filename("linear.txt"),
        names="linear1",
        pattern=list("x[i]",
                     "a  ~  dnorm(0,0.0001)"
                     ),
        replacement=list("x[i] + pow(c,2)",
                      "a  ~  dunif(0,10000) \n c  ~  dnorm(0,0.0001)
                          "
                      ),
        out=".",
```

```
          suffix = ".txt"
          )
```

The `replacement` argument will also accept rows or columns from a dataframe. Using this approach, a set of initial estimates could be generated in R then passed to `metaSub` to populate specific "controller" file areas. In the case of NONMEM, this approach could be used to generate control streams for clinical trial simulation.

In addition to generating multiple "controller" files, there is a frequent need to generate permutations of the analysis data set. One typical scenario in population PK or PKPD analysis involves the generation of bootstrap data sets to use as input for the "final" model to aid in the generation of confidence intervals on the parameters. We often generate 500 to 1000 bootstrapped data sets using a resampling with replacement approach with some level of stratification on dose, covariates, etc ...The `resample()` function is a generic method with a class for dataframe that can be used for permutating a data set. Lets take a look at the arguments for `resample`.

Listing 8.10:

```
args(resample.data.frame)
```

```
function (x, names, key = NULL, rekey = FALSE, out = NULL, stratify
    = NULL,
    ext = ".csv", row.names = FALSE, quote = FALSE, sep = ",",
    replace = TRUE, ...)
NULL
```

For our example, we are interested in generating 20 additional data sets resampled on "ID" and stratified on "weight" and "sex" using the `prob1` dataframe. In the function call below, "ID" is the key that identifies unique individuals, "rekey" = TRUE tells resample to generate a new "ID" variable in each data set starting from 1, "names" indicates the naming convention for the output *.csv files, and "out" controls the location of the new data sets. If "out" is not specified, the resampled data sets are combined into a list with elements corresponding to "names".

Listing 8.11:

```
prob1<-read.table(file="prob1.tab", skip=0, header=TRUE)
head(prob1)
```

```
  C ID    DV AMT II ADDL TIME RATE  HT WT   CLCR SEX AGE
1 0  1  0.00 900 12    9  0.0  300 103 91 101.48   0  46
2 0  1  6.99   0  0    0  1.5    0 103 91 101.48   0  46
3 0  1 12.66   0  0    0  3.0    0 103 91 101.48   0  46
4 0  1  5.31   0  0    0  8.0    0 103 91 101.48   0  46
5 0  1  2.39   0  0    0 11.9    0 103 91 101.48   0  46
6 0  1  2.87   0  0    0 23.9    0 103 91 101.48   0  46
```

Listing 8.12:

```
summary(prob1$WT)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  34.00   50.50   58.00   60.52   70.25   96.00
```

Listing 8.13:

```
resample(
         prob1,
         key="ID",
         rekey=TRUE,
         names=1:20,
         out=".",
         stratify=list(prob1$WT>60, prob1$SEX)
         )
```

Each of the resulting data sets will contain the same number of subjects (n=40), a similar split of subjects less than 60 kg, and a similar number of males and females in each weight grouping. Due to resampling with replacement, some of the individuals may be present in each data set more than once or not at all.

8.5 Reading and plotting NONMEM output

For every model that is run in NONMEM, there are some typical diagnostic plots that are evaluated. These include Predicted and Individual Predicted vs Observed and Weighted/-Conditional Weighted Residuals vs Predictions and Time. If `NONR72()` is used to run NONMEM these plots are generated following every run by a call to the `PLOTR()` function. Additional arguments to `NONR72()` can be provided to evaluate the inter-individual random effects, categorical and continuous covariates, and the relationship between the inter-individual random effects and covariates. An example of a `NONR72` call that demonstrates the full plotting capabilities of `PLOTR()` is shown below.

Listing 8.14:

```
NONR(project = "/home/billk/MI205/modelingoutr/",
     command="/opt/NONMEM/nm72gf/nmqual/autolog.pl",
     run = c(4),
     grid=TRUE,
     cont.cov = c("WT","AGE", "CLCR"),
     cat.cov = c("SEX"),
     eta.list = c("ETA1","ETA2","ETA3"),
     par.list = c("CL","V"),
     grp = c("SEX"),
```

```
      dvname = "Test ng/mL"
      )
```

A pdf file (DiagnosticPlotReview_4.pdf) has been generated containing all of the diagnostic and covariate plots. Since we specified the "grp" argument the diagnostic portion of the plots were generated twice, once for each level of the "grp" variable. The plots created by `NONR72()` and `PLOTR()` provide a set of standardized plots for evaluating models. In many situations, there are other plots that the pharmacometrician would like to evaluate for each or at least for a subset of runs. The `epilog` argument to `NONR72()` will run a user defined R script following completion of the standard plots. The script "epilogEx.R" is distributed with the `metrumrg` package as an example. A `NONR72()` call implementing an epilog script is shown below. (Lets take a quick look at the "epilogEx.R" script.)

Listing 8.15:

```
NONR(project = "/home/billk/MI205/modelingoutr/",
     command="/opt/NONMEM/nm72gf/nmqual/autolog.pl",
     run = c(4),
     grid=TRUE,
     cont.cov = c("WT","AGE", "CLCR"),
     cat.cov = c("SEX"),
     eta.list = c("ETA1","ETA2","ETA3"),
     par.list = c("CL","V"),
     grp = c("SEX"),
     dvname = "Test ng/mL",
     epilog = paste(" ~ /MI205/modelingoutr/epilogEx.R")
     )
```

The `epilog` script is not limited to plotting functionality. Any type of post-run analysis, plotting, data formatting, etc...can be performed with the script. This approach allows for the automatic generation of additional plots or analysis after the completion of a NONMEM run and provides access to a few variables that would not typically be available outside of the `NONR72()` call.

8.6 Reading and plotting WinBUGS output

The application of Bayesian analysis in the field of pharmacometrics is growing and is a tool being applied to aid in decision making during the drug development process. The main tool used in pharmacometrics for Bayesian analysis is WinBUGS (http://www.mrc-bsu.cam.ac.uk/bugs/winbugs/contents.shtml. We will limit todays information to reading and plotting WinBUGS output. (Those interested in the application of Bayesian analysis can review the WinBUGS webpage or the Metrum Institute (http://www.metruminstitute.org/ course offerings.)

For our example today, we will be using the WinBUGS output from a simple linear regression model. The model is described below.

$$y_i = a + b * x_i + \sigma_i \tag{8.1}$$

The output was saved to a file (linear1b.fit.save) using `save()` and we will load it back into R using `load()`. During the actual WinBUGS run the parameters that were monitored were a, b, and sigma. A set of functions found in the "bugs.tools.R" script were used to run WinBUGS, save the model output, and format the data for plotting. This script is part of the bugsParallel (http://code.google.com/p/bugsparallel) project on GoogleCode and has also been placed in the directory of files for this lecture.

Listing 8.16:

```
load("~/MI205/modelingoutr/linear/linear1b/linear1b.fit.Rsave")
```

There is a new R object "bugs.fit" that contains all of the information from the WinBUGS run as a `list`. We can generate some diagnostic plots and a table of the output using the code below.

Listing 8.17:

```
courseDir = "."
model.name = "linear1b" # root names of model file
example.dir = "linear" # subdirectory containing model file
setwd(courseDir)
library(coda)
library(lattice)
source("bugs.tools.R")
# create WinBUGS data set
bugs.data = list(
       x = c(1,2,3,4,5,6,7,8,9,10),
       y = c(5.19,6.56,9.19,8.09,7.6,7.08,6.74,9.3,8.98,11.5)
)
# specify the variables for which you want history and density
   plots
parameters.to.plot =c("a","b","sigma")
#####################################################
# rename and reformat MCMC results
# to facilitate later calculations and plots

sims.array = bugs.fit$sims.array
posterior = array(as.vector(sims.array),dim=c(prod(dim(sims.array)
   [1:2]),dim(sims.array)[3]),
       dimnames=list(NULL,dimnames(sims.array)[[3]]))
#####################################################
```

```
# posterior distributions of parameters

# open graphics device
pdf(file = paste(courseDir,"/",example.dir,"/",model.name,"/",
   model.name,".plots.pdf",sep=""),width=6,height=6)
# subset of sims.array containing selected variables
x1 = sims.array[,,unlist(sapply(c(paste("^",parameters.to.plot,"$
   ",sep=""),
       paste("^",parameters.to.plot,"\\[",sep="")),grep,x=dimnames
          (sims.array)[[3]]))]
# create history, density and Gelman-Rubin-Brooks plots, and a
   table of summary stats
ptable = parameter.plot.table(x1)
write.csv(signif(ptable,3),paste(example.dir,"/",model.name,"/",
   model.name,".summary.csv",sep=""))
#######################################################
# posterior predictive distributions

pred = posterior[,grep("ypred\\[",dimnames(posterior)[[2]])]
x1 = data.frame(x=bugs.data$x,y=bugs.data$y)
x1$type =rep("observed",nrow(x1))
x2 = rbind(x1,x1,x1)
x2$y = as.vector(t(apply(pred,2,quantile,probs=c(0.05,0.5,0.95))))
x2$type = rep(c("5%ile","median","95%ile"),ea=nrow(x1))
x1 = rbind(x1,x2)
xyplot(y ~ x,x1,groups=type,panel=panel.superpose,
       type=c("l","l","l","p"),lty=c(3,3,1,0),col=c(2,2,4,1),
       pch=c(NA,NA,NA,19),cex=1.5,lwd=3,
       scales=list(cex=1),xlab=list(xlab="x",cex=1.2),
       ylab=list(ylab="y",cex=1.2))
dev.off()
```

The plots can be found in the "linear1b" directory as a pdf (linear.1b.plots.pdf) and the summary table of the model parameters as a csv file (linear.1b.summary.csv).

8.7 Summarizing a set of NONMEM runs

Following a series of NONMEM runs, it is convenient to view all of the parameter estimates in one file. The `metrumrg` package contains a function called `rlog()` that will generate a *.csv file containing the relevant results of each completed run. The arguments to `rlog` are a subset of those required for `NONR72`.

Listing 8.18:

```
args(rlog)
```

```
function (run, project = getwd(), boot = FALSE, append = TRUE,
    tool = "nm6", file = filename(project, "CombRunLog.csv"),
    rundir = filename(project, run, if (boot) ".boot" else ""),
    nmlog = file.path(rundir, "NonmemRunLog.csv"), nmout = filename
        (rundir,
         run, ".lst"), pattern = c("^F[ISRCMP]", "^OU", "^nonmem",
         "^nul$", "WK", "LNK$", "fort", "^nm", "lnk$", "set$",
         "^gar", "INT", "^temp", "^tr", "^new", "^FD", "^Run\\d+\\.o
            \\d+$",
         "^prsizes"), ...)
NULL
```

To generate a summary of the two runs (1 and 4) we performed earlier in class we could use the following code.

Listing 8.19:

```
rlog(project="/home/billk/MI205/modelingoutr.",
     run=c(1,4),
     tool="nm7",
     append=FALSE)
```

`rlog` reads in a subset of output files created in the run directory for each run when `NONR72()` is used to run NONMEM and collates the results into one file. It is best to view the resulting file (CombRunLog.csv) in a spreadsheet-type (MSExcel, Numbers, etc. . .) program. This file can also be used to generate the "Model Building Summary Table" that is usually included as part of a population PK or PKPD report.

8.8 Homework

- In the `metaSub` example 10 new control streams (100.ctl - 110.ctl) were generated from 1.ctl. How could this code be changed to generate 100 control streams using numbering that starts from 200? Would it be possible to number the control streams by starting from 200 but use only even numbers? If so, how might we do it?
- Use `resample()` to generate 10 new versions of the `prob1` data set. Plot a histogram of WT for the original data and for each of the resampled data sets. Does the distribution of WT in the resampled data sets appear similar to the original?

- Using the same scenario, generate the resampled data sets and the plots of the resampled data sets in two function calls. HINT: One call to `resample()` and one call to `lapply`.
- Read in output and generate 3 - 4 diagnostic plots for any modeling performed outside of R. (You do not need to share this with the class.) What problems, if any, did you encounter? What types of plots did you create?

Chapter 9

Advanced Function Writing

9.1 Objectives

After completing this chapter, you will be able to ...

- Locate errors in the execution of a function.
- Test assumptions about supplied arguments.
- Provide performance information to the user.
- Control function behavior conditionally.
- Manage repetition of function steps.
- Override default print behaviors.
- Implement new methods for generic functions.

9.2 Introduction

Writing useful functions contributes enormously to the power, convenience, and elegance of R (R help, Introduction to R, Section 10). Chapter 4 above focused on the externals of function writing: names, arguments, storage, etc. This chapter focuses on the internals. In general, we examine techniques for refining the function body for better performance. We also take a first look at the power of object orientation.

9.3 Debugging

You won't get very far writing your own functions if you don't have a debugging strategy. The problem is that, even if a function is syntactically correct, many kinds of errors can occur at runtime (during function execution). These include programming errors as well as unanticipated characteristics of the supplied arguments. It is essential to be able to examine errors as they occur.

The strategy used here emphasizes `traceback()`, `debug()`, and to some extent `browser()`. Other strategies exist (see, for example, the CRAN package `debug`). The function `traceback()` generally gives us a good idea of where things went wrong. `debug()` lets us step through a named function line-by-line, examining inputs and outputs. `browser()` does something similar for anonymous functions.

As you know, functions can call functions, which call other functions, and so on, creating a 'stack' of functions that all contribute to a single result. `traceback()`, typed on the command line, shows us the stack of functions that were active at the time the last error occured.

To illustrate, let's create a system of toy functions for calculating root mean squared error.

Listing 9.1:

```
deviation <- function(x)x-mean(x)
squared <- function(x)x**2
root <- function(x)x**0.5
values <- c(1,'NA',3,4,5)
root(mean(squared(deviation(values))))
```

```
Error in x - mean(x) : non-numeric argument to binary operator
In addition: Warning message:
In mean.default(x) : argument is not numeric or logical: returning
   NA
```

In this case, its fairly obvious where the error occured. To verify, we type on the command line:

Listing 9.2:

```
traceback()
```

```
4: deviation(values)
3: squared(deviation(values))
2: mean(squared(deviation(values)))
1: root(mean(squared(deviation(values))))
```

So, `root()` called `mean()`, `mean()` called `squared()`, `squared()` called `deviation()`, and `deviation()` gave an error. We have the complete pathway back to the function in which the error occured.

Now we wish to explore the problem in greater detail. What was the value of `x` at the time of the error? We issue the command `debug(deviation)`, and then recreate the problem. When `deviation()` is called, execution pauses, and we get a command prompt. We can execute arbitrary expressions, or we can hit `return` to step through the code one line at a time, or `Q` to quit.

Listing 9.3:

```
debug(deviation)
root(mean(squared(deviation(values))))

debugging in: deviation(values)
debug: x - mean(x)
```

Listing 9.4:

```
Browse[1]>ls()

[1] "x"
```

Listing 9.5:

```
Browse[1]>x

[1] "1"  "NA" "3"  "4"  "5"
```

Listing 9.6:

```
Browse[1]>mean(x)

[1] NA
Warning message:
In mean.default(x) : argument is not numeric or logical: returning
   NA
```

Listing 9.7:

```
Browse[1]>class(mean(x))

[1] "numeric"
Warning message:
In mean.default(x) : argument is not numeric or logical: returning
   NA
```

Listing 9.8:

```
Browse[1]>class(x)
```

```
[1] character
```

When we evaluate `mean(x)`, we get a warning, because `x` is `character`. Even though `mean()` doesn't like getting character, it returns a *numeric* NA, which *is* a legal argument to the binary operator `-`. The *real* problem is `x` itself, which is *not* a legal argument for subtraction.

At this point, we enter `Q` to quit, and `undebug(deviation)` to turn off debugging on that function (debugging turns off automatically if we redefine the function).

9.4 Testing Input

Now that we understand the problem, we can improve our function to reduce (unexplained) errors. Specfically, we can test each argument to make sure it corresponds to our expectations. We'll try to do something intellligent if it does not.

Listing 9.9:

```
deviation <- function(x){
  if(!is.numeric(x))return(NULL)
  return(x-mean(x))
}
root(mean(squared(deviation(values))))
```

```
NaN
```

This time, we have trapped the condition where `x` is `character`, rather than simply letting an error occur. Now we fix our data, and try again.

Listing 9.10:

```
values <- c(1,NA,3,4,5)
root(mean(squared(deviation(values))))
```

```
NA
```

No complaints here, but the NA is still preventing a defined result.

9.5 Alerting the User

What started out looking like a relatively simple problem is getting complicated. Maybe its time to throw some of the burden back on the user. There are three communication functions, analogous to green, yellow, and red traffic signals: `message()` provides neutral information; `warning()` tells the user that a problem may exist; `stop()` signals an error and halts function execution.

Listing 9.11:

```
deviation <- function(x){
  message('calculating differences from the reference value ...')
  if(any(is.na(x)))warning('removing NA values in x')
  if(!is.numeric(x))stop('x must be numeric')
  return(x-mean(x,na.rm=TRUE))
}
```

Listing 9.12:

```
deviation(1:5)
```

```
calculating differences from the reference value ...
[1] -2 -1  0  1  2
```

Listing 9.13:

```
deviation(c(1,NA,2,3,4,5))
```

```
calculating differences from the reference value ...
[1] -2 NA -1  0  1  2
Warning message:
In deviation(c(1, NA, 2, 3, 4, 5)) : removing NA values in x
```

Listing 9.14:

```
deviation(c(1,'NA',2,3,4,5))
```

```
calculating differences from the reference value ...
Error in deviation(c(1, "NA", 2, 3, 4, 5)) : x must be numeric
```

9.6 Branching

A program branches when it does one of several possible things, depending on some condition. Above we branched to a warning when an NA was detected, using the familiar `if()`

syntax. Like many languages, R also supports the use of `else` to handle instances not matching the condition passed to `if()`. Let's revisit that example, but this time make sure that our function gives a message or a warning or an error, but never two of these.

Listing 9.15:

```
deviation <- function(x){
  if(!is.numeric(x))stop('x must be numeric')
  else if(any(is.na(x)))warning('removing NA values in x')
  else message('calculating differences from the reference value
     ...')
  return(x-mean(x,na.rm=TRUE))
}
deviation(c(1,NA,2,3,4,5))
```

```
[1] -2 NA -1  0  1  2
Warning message:
In deviation(c(1, NA, 2, 3, 4, 5)) : removing NA values in x
```

Note that the neutral message is not printed this time.

As illustrated, conditions can be chained together with `else if`.

Do not confuse the `if...else` construct with the function `ifelse()`, which has a very different purpose. In particular, `ifelse()` accepts a *vector* (any length) of `TRUE`s and `FALSE`s, whereas `if()` always takes exactly one expression evaluating to `TRUE` or `FALSE`.

By the way, this was a fairly simple example. For something a little more complicated, curly braces could be used to group the statements that follow `if()` or `else`.

9.7 Looping

R has syntactical features that support evaluating (sets of) expressions repeatedly. These include `for()`, `repeat`, and `while()`. By far the most common is `for()`; the others will not be further discussed here.

`for()` iterates across a sequence of values, evaluating the same expressions for each value in turn. The value itself is available to the expressions, and can be used to influence the result.

Note that the use of `for()` is often discouraged in favor of `sapply()` or `lapply()`. In comparison, `for()` is much easier to understand, but is also *much* slower. Generally you should prefer an `apply` solution if there is a straight-forward way to write one, especially when writing functions that will be used a lot. In some cases though, the extra time you spend

writing may be greater than the time you save running the the function! Furthermore, `for()` has that iteration-specific value for guiding outcomes: hard to emulate with `apply()` solutions.

To illustrate, we'll solve the same problem both ways. We want a function that accepts a data.frame and returns a logical vector telling whether each column is numeric.

Listing 9.16:

```
numerics <- function(x){
  result <- logical(ncol(x))
  for(col in 1:ncol(x)){
    result[[col]] <- is.numeric(x[[col]])
    names(result)[[col]] <- names(x)[[col]]
  }
  return(result)
}
numerics(Theoph)
```

```
Subject      Wt    Dose    Time    conc
  FALSE    TRUE    TRUE    TRUE    TRUE
```

Listing 9.17:

```
numerics <- function(x)sapply(x,is.numeric)
numerics(Theoph)
```

```
Subject      Wt    Dose    Time    conc
  FALSE    TRUE    TRUE    TRUE    TRUE
```

Even though the first example has 8 times as many lines of code, one might argue that it's easier to understand. `col` takes on the values 1 through 5, one at a time, and does something explicit with them. The second example is elegant and takes about 1/7th the time to run. But you may have to spend some time to understand it if you're not already familiar with `sapply`.

9.8 Controling Visibility

Expressions evaluated on the command line are implicitly printed.

Listing 9.18:

```
names(Theoph[1])
```

```
[1] "Subject"
```

Inside `for()` loops and functions, however, implicit printing is turned off.

Listing 9.19:

```
dfnames <- function(x)for(name in names(x))name
dfnames(Theoph)
```

No visible result. To force visibility within a function or `for()` loop, we need to print explicitly.

Listing 9.20:

```
dfnames <- function(x)for(name in names(x))print(name)
dfnames(Theoph)
```

```
[1] "Subject"
[1] "Wt"
[1] "Dose"
[1] "Time"
[1] "conc"
```

`lapply()` has the opposite problem: sometimes it leaves a noisy command line record, where silence was preferred. Suppose we want to unpack a `data.frame` by assigning each column as a separate object in the global environment.

Listing 9.21:

```
unpack <- function(x)lapply(
    names(x),
    function(name)assign(
        name,
        x[[name]],
        pos=1
    )
)
unpack(Theoph[1:2,])
```

```
[[1]]
[1] 1 1
12 Levels: 6 < 7 < 8 < 11 < 3 < 2 < 4 < 9 < 12 < ... < 5

[[2]]
[1] 79.6 79.6

[[3]]
[1] 4.02 4.02

[[4]]
```

```
[1] 0.00 0.25

[[5]]
[1] 0.74 2.84
```

Now, the intent was to create 'column' objects in the global environment, not to print all that data on the console. to suppress unwanted printing, we wrap the return value (in this case, the whole function body) in a call to `invisible()`.

Listing 9.22:

```
unpack <- function(x)invisible(
    lapply(
        names(x),
        function(name)assign(
            name,
            x[[name]],
            pos=1
        )
    )
)
unpack(Theoph[1:2,])
```

This time, silence! (But take my word for it that the global objects were created.)

9.9 Writing Methods

9.9.1 Background

Many modern computer languages support *object orientation*. Indeed, some (like Java and C++) are completely object-oriented. Others, (like Perl and R) are accommodating, but support older programming paradigms as well.

Object orientation means that the result of an action depends on the type of thing acted on. In R, an example of an action is the `print()` function, and "the type of thing" could mean the class of an argument. In object-oriented languages, data and procedures are tightly connected.

Object orientation also means that properties of things can be hierarchically arranged. In R, for example, everything is an object, and everything has the property `length`. But data frames have more properties than that (such as `names`) and special kinds of data frames have even more.

R has two systems for implementing object-oriented behavior. S3 is older, easy to learn, but somewhat brittle (difficult to use for complex tasks). S4 is more modern and robust, but rather hard to learn. Even though S4 is "recommended for new projects", S3 is likely not going away anytime soon, since much of the base behavior of R continues to rely on it. Here we will use S3.

Now for the "all around you" monologue: in R, object orientation is all around you. You use it all the time without thinking about it. Consider the `summary` function. When you summarize something, R tries to find a *method* appropriate for the class of that thing. The `summary` function is called a *generic* function; its main purpose is to trigger the search for an appropriate `method`.

Listing 9.23:

```
summary
```

```
function (object, ...)
UseMethod("summary")
<environment: namespace:base>
```

`UseMethod()` does the magic. When `x` is a `data.frame`, it searches for a function called `summary.data.frame()` and redirects your request to that, if it exists. Actually, the class of an object is a `character` vector of arbitrary length; `UseMethod()` will try all the elements in the class vector until it finds something. If it finds nothing, it will look one more time for a default method, e.g. `summary.default()`.

So, what specific methods have been defined for the generic function `summary`?

Listing 9.24:

```
methods(summary)
```

```
 [1] summary.Date            summary.POSIXct
 [3] summary.POSIXlt         summary.aov
 [5] summary.aovlist         summary.aspell*
 [7] summary.connection      summary.data.frame
 [9] summary.default         summary.ecdf*
[11] summary.factor          summary.glm
[13] summary.infl            summary.lm
[15] summary.loess*          summary.manova
[17] summary.matrix          summary.mlm
[19] summary.nls*            summary.packageStatus*
[21] summary.ppr*            summary.prcomp*
[23] summary.princomp*       summary.srcfile
[25] summary.srcref          summary.stepfun
[27] summary.stl*            summary.table
[29] summary.tukeysmooth*
```

```
   Non-visible functions are asterisked
```

Note the method `summary.data.frame()`. Recalling the tight connection in object orientation between data and procedures, we can invert the previous question and ask: what specific methods have been defined for the class `data.frame`?

Listing 9.25:

```
methods(class='data.frame')
```

```
 [1] $<-.data.frame           Math.data.frame
 [3] Ops.data.frame           Summary.data.frame
 [5] [.data.frame             [<-.data.frame
 [7] [[.data.frame            [[<-.data.frame
 [9] aggregate.data.frame     anyDuplicated.data.frame
[11] as.data.frame.data.frame as.list.data.frame
[13] as.matrix.data.frame     by.data.frame
[15] cbind.data.frame         dim.data.frame
[17] dimnames.data.frame      dimnames<-.data.frame
[19] droplevels.data.frame    duplicated.data.frame
[21] edit.data.frame*         format.data.frame
[23] formula.data.frame*      head.data.frame*
[25] is.na.data.frame         mean.data.frame
[27] merge.data.frame         na.exclude.data.frame*
[29] na.omit.data.frame*      plot.data.frame*
[31] print.data.frame         prompt.data.frame*
[33] rbind.data.frame         row.names.data.frame
[35] row.names<-.data.frame   rowsum.data.frame
[37] split.data.frame         split<-.data.frame
[39] stack.data.frame*        str.data.frame*
[41] subset.data.frame        summary.data.frame
[43] t.data.frame             tail.data.frame*
[45] transform.data.frame     unique.data.frame
[47] unstack.data.frame*      within.data.frame

   Non-visible functions are asterisked
```

In other words, all those generics (`aggregate`, `by`, `dim`, `edit`, ...) have special meaning for data frames.

9.9.2 Implementation

How can we take advantage of the S3 mechanism to achieve desired results? The simplest strategy is:

1. Define a new method for an existing generic and class.

For example, no one has defined a method for generic `summary` and class `numeric`. We could create `summary.numeric()`, and then all numerics everywhere would be forced to summarize as we specify.

Despite the intoxicating attraction of its raw power, we might rightly guess that such an approach could have unintended consequences. A slightly more sophisticated (and safer) approach:

1. Define a new class.
2. Define a new method for an existing generic and the new class.

We'll try this later. If we were really ambitious (not today), we would:

1. Define a new class.
2. Define a new generic.
3. Define a new method for the new generic and the new class.

Feel free to try that on your own!

For now, let's try the second approach. Suppose we wish to have a more parametric statistical summary of `conc` column in `Theoph`. First, we reclassify `conc`.

Listing 9.26:

```
class(Theoph$conc) <- 'parametric'
```

Next, we write the appropriate method for `parametric`.

Listing 9.27:

```
summary.parametric <- function(x,...)c(
    mean=mean(x),
    var=var(x),
    std=sd(x),
    lo=mean(x)-1.96*sd(x),
    hi=mean(x)+1.96*sd(x)
)
```

Now `conc` will summarize as desired.

Listing 9.28:

```
summary(Theoph$conc)
```

```
     mean        var        std         lo         hi
4.9604545  8.2215204  2.8673194 -0.6594914 10.5804005
```

Listing 9.29:

```
summary(Theoph)
```

```
   Subject          Wt              Dose
6      :11   Min.   :54.60   Min.   :3.100
7      :11   1st Qu.:63.58   1st Qu.:4.305
8      :11   Median :70.50   Median :4.530
11     :11   Mean   :69.58   Mean   :4.626
3      :11   3rd Qu.:74.42   3rd Qu.:5.037
2      :11   Max.   :86.40   Max.   :5.860
(Other):66
     Time           conc
Min.   : 0.000   mean: 4.9605
1st Qu.: 0.595   var : 8.2215
Median : 3.530   std : 2.8673
Mean   : 5.895   lo  :-0.6595
3rd Qu.: 9.000   hi  :10.5804
Max.   :24.650
```

9.10 Exercises

1. This function sorts a data frame by the levels in `index`. Use `traceback()` and `debug()` to identify why the second example doesn't work.

Listing 9.30:

```
sort.my <- function(x,index){
       index <- as.factor(index)
       order <- order(index)
       x <- x[order,]
       return(x)
}
sort.my(Theoph,Theoph$Subject)
sort.my(Theoph[,'conc'],Theoph$Subject)
```

2. Modify `sort.my()` so that it sends a message if x is a `matrix`, warns if index is not as long as `nrow(x)` and stops if `x` is not a `data.frame` or `matrix`.
3. Modify `sort.my()` so that if `index` is a length-one `character` vector, it is assumed to be a column name in `x`.
4. Modify `sort.my()` so that it rounds every `numeric` column to 2 significant digits.
5. Modify `sort.my()` so that the result is invisible. What use is an invisible result?

6. Reclassify `Theoph` as `c('my','data.frame')` and sort it using a generic function.

Chapter 10

Advanced Exercises

10.1 Objectives

After completing this chapter, you will be able to . . .

- Understand the usage of `do.call()`
- Understand the usage of `ifelse()`
- Understand the usage of `all()` and `any()`
- Gain an understanding of what R GUI's are available and their pros and cons

10.2 Introduction

Up to this point, the chapters have followed a weekly theme. In this chapter, we will move away from that approach and explore a set of functions that have general applicability across plotting, data formatting, and function writing.

10.3 The use of `ifelse()`

We will start this exploration with the `ifelse()` function. `ifelse()` is used for "conditional" element selection. The set of arguments are limited to three.

Listing 10.1:

```
args(ifelse)
```

```
function (test, yes, no)
NULL
```

"test" represents the vector and condition that will be evaluated, "yes" is the return values if "test" is TRUE and "no" is the return value if "test" is FALSE. A simple example of `ifelse()` is shown below.

Listing 10.2:

```
a <- c(1,2,3,-3,-4,4,6,7)
sqrt(a) # this results in a warning message for the negative
   values
```

```
[1] 1.000000 1.414214 1.732051      NaN      NaN 2.000000
[7] 2.449490 2.645751
```

What if we were aware of the negative values and wanted to calculate the `sqrt(a)` without getting warning methods or producing `NaN` values? Lets conditionally test "a" for negative values and evaluate only those values of "a" that are positive.

Listing 10.3:

```
sqrt(ifelse(a > 0, a, NA)) # no warnings
```

```
[1] 1.000000 1.414214 1.732051       NA       NA 2.000000
[7] 2.449490 2.645751
```

The example above could likely be executed in other ways. However, one of the strengths of `ifelse()` is the ability to string together nested calls to `ifelse()` to create a new variable. We will create a data.frame that we can use to demonstrate the functionality of `ifelse()`.

Listing 10.4:

```
set.seed(21345)
tage <- data.frame("AGE"=runif(100, 18, 45),
                   "WT"=runif(100,40,100),
                   "SEX"=c(rep(0,50),rep(1,50)),
                   # 0 = female, 1 = male
                   "SCR"=c(runif(100,0.5,1.8))
                   )
summary(tage)
```

```
      AGE              WT              SEX
 Min.   :18.70   Min.   :40.53   Min.   :0.0
 1st Qu.:24.07   1st Qu.:53.15   1st Qu.:0.0
 Median :31.11   Median :68.76   Median :0.5
 Mean   :30.99   Mean   :70.14   Mean   :0.5
```

```
 3rd Qu.:37.88   3rd Qu.:86.11   3rd Qu.:1.0
 Max.   :44.91   Max.   :98.59   Max.   :1.0
      SCR
 Min.   :0.5213
 1st Qu.:0.8453
 Median :1.2370
 Mean   :1.1995
 3rd Qu.:1.5736
 Max.   :1.7927
```

Now, we can use `ifelse()` to create a new variable that will create "bins" of age.

Listing 10.5:

```
tage$bin <- ifelse(tage$AGE < 20, 1,
                   ifelse(tage$AGE < 30, 2,
                          ifelse(tage$AGE < 40, 3,
                                 4
                                 )
                          )
                   )
head(tage, 10)
```

```
        AGE       WT SEX       SCR bin
1  27.39694 80.64919   0 0.7656902   2
2  44.09735 87.38638   0 1.7577789   4
3  27.34969 95.57867   0 1.6070393   2
4  39.11794 83.00141   0 1.2589200   3
5  26.96996 67.93889   0 0.6574091   2
6  28.58650 59.40342   0 0.6801658   2
7  19.19439 94.04131   0 1.2084703   1
8  27.64331 49.65341   0 1.2464154   2
9  23.33059 62.84055   0 1.7408042   2
10 40.11902 47.78716   0 1.7566697   4
```

The use of nested `ifelse()` commands allows for the creation of unique age bins based on the results of the "test" argument. This can be contrasted to the use of multiple subset arguments based on the value of AGE that will not provide the same results. For example,

Listing 10.6:

```
tage$bin2[tage$AGE<20] <- 1
tage$bin2[tage$AGE<30] <- 2
tage$bin2[tage$AGE<40] <- 3
head(tage, 10)
```

```
        AGE       WT SEX       SCR bin bin2
1  27.39694 80.64919   0 0.7656902   2    3
2  44.09735 87.38638   0 1.7577789   4   NA
3  27.34969 95.57867   0 1.6070393   2    3
4  39.11794 83.00141   0 1.2589200   3    3
5  26.96996 67.93889   0 0.6574091   2    3
6  28.58650 59.40342   0 0.6801658   2    3
7  19.19439 94.04131   0 1.2084703   1    3
8  27.64331 49.65341   0 1.2464154   2    3
9  23.33059 62.84055   0 1.7408042   2    3
10 40.11902 47.78716   0 1.7566697   4   NA
```

We can compare the results in `tage$bin2` to `tage$bin` and demonstrate that the results are very different. The assignments in "bin2" are not correct because each row is not "conditioned" on the "test" executed in the previous row. We could also use `ifelse()` to conditionally apply a function. We can demonstrate this by calculating creatine clearance (CRCL). As a reminder, CRCL is based on weight, age, sex, and serum creatinine using the following equation.

$$CRCL(\text{ml/min}) = ((140 - \text{Age(yr)}) * \text{weight(kg)})/(\text{serumcreatinine} * 72) * 0.85 \text{forfemales} \tag{10.1}$$

Given the equation above, we can write an R function then use `ifelse` to apply it conditionally to males and females.

Listing 10.7:

```
crclx <- function(age, wt, scr){
 tmp <- (140-age)*wt/(scr*72)
 return(tmp)
}
tage$CRCL <- ifelse(tage$SEX==1,
                    crclx(tage$AGE, tage$WT, tage$SCR),
                    crclx(tage$AGE, tage$WT, tage$SCR) * 0.85
                    )
head(tage)
```

```
       AGE       WT SEX       SCR bin bin2      CRCL
1 27.39694 80.64919   0 0.7656902   2    3 140.01790
2 44.09735 87.38638   0 1.7577789   4   NA  56.28550
3 27.34969 95.57867   0 1.6070393   2    3  79.09578
4 39.11794 83.00141   0 1.2589200   3    3  78.52134
5 26.96996 67.93889   0 0.6574091   2    3 137.89961
6 28.58650 59.40342   0 0.6801658   2    3 114.87378
```

It is important to differentiate between `ifelse` and the `if()` and `else()`. The latter two are not really "functions" in the manner that we have been describing "functions". They are control-flow constructs of the R language. (See Chapter 9 for a demonstration of the use of `if` and `else`.) One obvious difference between `ifelse()` and `if` is that `if` expects a length-one logical vector but `ifelse()` can read and act on a vector of any length. NOTE: If a vector of length greater than one is supplied to `if()` a warning is generated, only the first element is used, and the code block continues. This can and often will result in unexpected outcomes.

10.3.1 Exercises

- Use `ifelse` to create a new variable in the `tage` dataframe that codes `SEX` as M and F rather than 1 and 0.
- How would we need to change the `ifelse` call to calculate CRCL to utilize the revised SEX variable?

10.4 The use of `do.call()`

`do.call()` constructs and executes a function call from a name or a function and a list of arguments to be passed to it. The arguments for the function are taken from an object of mode "list". A simple example is the application of `rbind` to a list of results to "bind" the list into one large dataset. We will create the data then demonstrate the use of `do.call`.

Listing 10.8:

```
tauc <- data.frame("AGE"=runif(10000, 2, 12),
                   "AUC"=runif(10000,500,5000),
                   "TRIAL"=sort(rep(c(1:100),100))
                   )
tauc$bins <- ifelse(tauc$AGE < 4, 1,
                   ifelse(tauc$AGE < 6, 2,
                          ifelse(tauc$AGE < 9, 3,
                                 4
                                 )
                          )
                   )
head(tauc)
```

```
        AGE      AUC TRIAL bins
1 10.436888 1466.439     1    4
2  9.912768 2259.733     1    4
```

```
3  9.108265 2922.912     1    4
4 10.415998 4773.869     1    4
5  6.036194 3478.515     1    3
6  7.163238 1973.059     1    3
```

Listing 10.9:

```
table(tauc$bins)
```

```
   1    2    3    4
2111 1986 2942 2961
```

Listing 10.10:

```
sauc <- aggregate(tauc$AUC, list(TRIAL=tauc$TRIAL,AGEB=tauc$bins),
    median)
saucq <- tapply(sauc$x, list(sauc$AGEB), quantile, probs=c(0.025,
   0.5, 0.975))
# saucq is a list object containing the results of applying
   quantile by AGE bin to sauc
rsaucq <- do.call("rbind",saucq)
head(rsaucq)
```

```
      2.5%      50%    97.5%
1 1961.512 2693.799 3656.879
2 1712.728 2712.024 3788.206
3 2026.808 2762.735 3520.773
4 2061.857 2802.863 3456.310
```

Another example would be the use of `do.call` following `split`. For this example, we will use the `prob1` data set. We will subset `prob1` by "ID", perform some operations on the individual subsets, then use `do.call` and `rbind` to put the subsets back together into one dataframe.

Listing 10.11:

```
prob1<-read.table(file="prob1.tab", skip=0, header=TRUE)
head(prob1)
```

```
  C ID    DV AMT II ADDL TIME RATE  HT WT   CLCR SEX AGE
1 0  1  0.00 900 12    9  0.0  300 103 91 101.48   0  46
2 0  1  6.99   0  0    0  1.5    0 103 91 101.48   0  46
3 0  1 12.66   0  0    0  3.0    0 103 91 101.48   0  46
4 0  1  5.31   0  0    0  8.0    0 103 91 101.48   0  46
5 0  1  2.39   0  0    0 11.9    0 103 91 101.48   0  46
6 0  1  2.87   0  0    0 23.9    0 103 91 101.48   0  46
```

Listing 10.12:

```
prob1.ind <- split(prob1, prob1$ID)
# use lapply to calculate the median DV for each individual
# and return the results to a new list of dataframes called res
res <- lapply(prob1.ind, function(df){
  df$mDV <- median(df$DV, na.rm=TRUE)
  return(df)
  }
 )
# look at the first item in the res list
head(res[1])
```

```
$`1`
   C ID    DV AMT II ADDL  TIME RATE  HT WT   CLCR SEX AGE
1  0  1  0.00 900 12    9   0.0  300 103 91 101.48   0  46
2  0  1  6.99   0  0    0   1.5    0 103 91 101.48   0  46
3  0  1 12.66   0  0    0   3.0    0 103 91 101.48   0  46
4  0  1  5.31   0  0    0   8.0    0 103 91 101.48   0  46
5  0  1  2.39   0  0    0  11.9    0 103 91 101.48   0  46
6  0  1  2.87   0  0    0  23.9    0 103 91 101.48   0  46
7  0  1  2.91   0  0    0  35.9    0 103 91 101.48   0  46
8  0  1  1.36   0  0    0  47.9    0 103 91 101.48   0  46
9  0  1  2.34   0  0    0  71.9    0 103 91 101.48   0  46
10 0  1  2.13   0  0    0 107.9    0 103 91 101.48   0  46
11 0  1  4.97   0  0    0 109.5    0 103 91 101.48   0  46
12 0  1 10.46   0  0    0 111.0    0 103 91 101.48   0  46
13 0  1  5.85   0  0    0 116.0    0 103 91 101.48   0  46
14 0  1  2.13   0  0    0 119.9    0 103 91 101.48   0  46
15 0  1  0.05   0  0    0 144.0    0 103 91 101.48   0  46
    mDV
1  2.87
2  2.87
3  2.87
4  2.87
5  2.87
6  2.87
7  2.87
8  2.87
9  2.87
10 2.87
11 2.87
12 2.87
13 2.87
14 2.87
15 2.87
```

Listing 10.13:

```
# use do.call to rbind the list of dataframes into one dataframe
prob2 <- do.call("rbind", res)
head(prob2)
```

```
    C ID    DV AMT II ADDL TIME RATE  HT WT   CLCR SEX AGE
1.1 0  1  0.00 900 12    9  0.0  300 103 91 101.48   0  46
1.2 0  1  6.99   0  0    0  1.5    0 103 91 101.48   0  46
1.3 0  1 12.66   0  0    0  3.0    0 103 91 101.48   0  46
1.4 0  1  5.31   0  0    0  8.0    0 103 91 101.48   0  46
1.5 0  1  2.39   0  0    0 11.9    0 103 91 101.48   0  46
1.6 0  1  2.87   0  0    0 23.9    0 103 91 101.48   0  46
     mDV
1.1 2.87
1.2 2.87
1.3 2.87
1.4 2.87
1.5 2.87
1.6 2.87
```

In this second example, we see a very efficient way of operating on subsets of an original dataframe without looping or needing to formally track the individual dataframes. This approach could be used to perform multiple operations on a dataframe subset followed by binding back together.

10.4.1 Exercises

- Read in `data5.csv`, split by `SEX`, change the `AMT` to 800 if SEX==0 and 1000 if SEX==1, and combine with `do.call`
- Read in `test1.csv` and paste the columns together using `do.call`

10.5 Usage of `all()` and `any()`

In many cases, it can be very helpful to "test" on a vector or group of vectors prior to performing additional manipulations or calculations. The `all()` and `any()` functions are concise ways of performing these tests. `all` returns TRUE if all of the values in "x" are TRUE and FALSE if at least one of the values in "x" are false. Otherwise the valies is NA. `any` returns TRUE if at least one value is TRUE and FALSE if all of the values are FALSE. Lets say we want to test the "AMT" column in `prob1` prior to summarizing or performing additional data set manipulations.

Listing 10.14:

```
prob1<-read.table(file="prob1.tab", skip=0, header=TRUE)
# AMT should be equal to zero when DV > 0
any(prob1$AMT[prob1$DV>0]==0)
```

```
[1] TRUE
```

For `all`, we can demonstrate its use in the situation when we want to populate a dataframe (prob1) with a column from another dataframe (prob2). Lets say we have a column in `prob2` that contains systolic blood pressure (SBP) measured at the same time as the "DV" variable in `prob1`. If we can be assurred that the TIME and ID columns match in the two dataframes, we can simply add the SBP column to the `prob1`. We can test this belief using the `all` function.

Listing 10.15:

```
prob2 <- prob1
prob2$SBP <- runif(length(prob2$ID),120,160)
# test if ID and TIME are the same in both dataframes
all(prob1$ID==prob2$ID)
```

```
[1] TRUE
```

Listing 10.16:

```
all(prob1$TIME==prob2$TIME)
```

```
[1] TRUE
```

Listing 10.17:

```
# both results are TRUE so we can copy SBP over to prob1
prob1$SBP <- prob2$SBP
```

Since the result of `all` and `any` is a length-one logical vector, you could pass the results to `if` as part of a function call or as part of a simple branch in an R script. Lets use a data set similar to what we created earlier for the CRCL calculation to demonstrate an example of using these functions with an `if` statement.

Listing 10.18:

```
tage <- data.frame("AGE"=runif(100, 18, 45),
                   "WT"=runif(100,40,100),
                   "SEX"=c(rep(0,30),rep(1,50), rep(NA,20)),
                   # 0 = female, 1 = male
                   "SCR"=as.character(c(round(runif(75,0.5,1.8),2)
                      ,rep("NA",25)))
                   )
```

```
crclx <- function(age, wt, scr){
 tmp <- (140-age)*wt/(scr*72)
 return(tmp)
}
crclx(tage$AGE, tage$WT, tage$SCR)
# The above function results in an error because not all variables
    are
# numeric. Lets say we wanted to test for numeric values and
   report
# the problem rather than getting this error.
crclx2 <- function(age, wt, scr){
 if(any(!is.numeric(age)|!is.numeric(wt)|!is.numeric(scr)))stop('
    variables must be numeric')
 tmp <- (140-age)*wt/(scr*72)
 return(tmp)
}
crclx2(tage$AGE, tage$WT, tage$SCR)
str(tage)
tage$SCR <- as.numeric(as.character(tage$SCR))
crclx(tage$AGE, tage$WT, tage$SCR)
```

The above example demonstrates the use of `any` to performing some initial testing inside of a function and demonstrates a way to test for a variable type in a situation when we know a numeric is required for the successful completion of the function.

10.6 Using/Choosing an R graphical user interface (GUI)

R, like most statistical programming languages, was initially available only from the command line. In the last few years, a number of GUI or GUI-like interfaces have become available for R. The benefit of a true point-n-click GUI rapidly declines as an individual users skills increase. Users quickly realize that finer control is available when scripts are written versus being generated from a point-n-click interface. Since R is open-source, the development of point-n-click interfaces has been limited. (This is in contrast to the highy developed GUI available in S-PLUS.) With that being said, the GUI's available in R are less about point-n-click and more about a programming environment. Lets take a look at some of the available programming environments for R and the benefits and drawbacks of each.

1. R for windows offers an interface to the R command line.

   Benefits

   - available as part of the base R install

- offers line by line and code block submission to R
- allows for the easy installation of R packages

Drawbacks

- no syntax highlighting
- no ability or limited ability to add/customize features

2. R for Mac OS X offers a well developed interface to R.

   Benefits

   - available as part of the base R install
   - offers line by line and code block submission to R
   - allows for the easy installation of R packages
   - provides syntax highlighting
   - automatically indents function code each time a script is opened

   Drawbacks

   - no ability or limited ability to add/customize features

3. RStudio for windows/Mac OS X/linux offers a nice graphical interface to R.

   Benefits

   - freely available via web download
   - offers line by line and code block submission to R
   - allows for the easy installation of R packages
   - provides windows for viewing plots
   - offers a client/server version that can be run via web browser

   Drawbacks

   - no ability or limited ability to add/customize features
   - client/server version may not work with all browsers.

4. Emacs is the swiss army knife of text editors and offers a highly customizable interface, Emacs Speaks Statistics (ESS), to many statistical packages.

   Benefits

   - available on Windows, Mac OS X, linux, as well as many other operating systems

- can be used in graphical mode (point-n-click) as well as via a text based connection, like ssh
- highly customizable with many add-on packages
- provides syntax highlighting
- automatically indents function code
- provides a one-stop shop for programming environments (C++, R, Splus, SAS, WinBUGS, etc..)

Drawbacks

- highly customizable
- learning curve can be high when using text based connections for those used to a GUI environment

There are a number of other R GUI's available, including Tinn-R, JGR, SciViews-R and others. Additional information on R GUI's and the pros and cons can be found on the R GUI project page (http://www.sciviews.org/_rgui).

10.7 Homework

- Load the built in R data set `Indometh`, create a new variable called `WT` that contains one unique value per `Subject` (the `WT` variable should range from 50 - 100 kilograms) , and use `ifelse` to create a `WTbin` variable that is "1" for the lowest three weights and "2" for the highest three weights.
- Perform the same operation above but demonstrate the use of `split` and `do.call` (and any other necessary functions) during the process.
- Demonstrate the use of `do.call` to run make a call to `xyplot` using any data of your choosing. Hint: `do.call` expects exactly two arguments, a named function, and an associated `list` of arguments. For this exercise you will need to create a `list` of the arguments to be used by `xyplot`.

Chapter 11

Advanced Data Assembly

After completing this chapter, you will be able to . . .

- Extract information from a character column of irregular format.
- Impute a variable using stratified vertical information.
- Transform tables dynamically.
- Calculate variables by combining vertical and horizontal criteria.
- Interconvert row-level and column-level distinctions among table cells.

11.1 Introduction

An important step in most pharmacokinetic analyses is the preparation of the analysis data set. In Chapter 6, we studied the basic techniques of data assembly. Here, we'll examine some advanced techniques of data assembly and manipulation.

11.2 Irregular Extraction

Earlier we used `substr()` to extract specific sections of character vectors. Some character vectors, however, are rather irregular in format. Consider these subject identifiers, which are combinations of protocol names and subject numbers.

Listing 11.1:

```
id <- c('PROT300-1', 'PROT10-35', 'PROT5-281')
data.frame(id=id)
```

```
         id
1 PROT300-1
2 PROT10-35
3 PROT5-281
```

Even though these identifiers happen to be the same length, there's no simple way to extract, say, the protocol specifier using `substr()`. Fortunately, there is a dash that separates the protocol name from the subject number. We can use it to isolate the information of interest, using `strsplit()`.

Listing 11.2:

```
splits <- strsplit(id,'-', fixed=TRUE)
splits
```

```
[[1]]
[1] "PROT300" "1"

[[2]]
[1] "PROT10" "35"

[[3]]
[1] "PROT5" "281"
```

Note that `strsplit()` returns a list of vectors. We can extract protocol names for each row using `sapply()` and the subset operator.

Listing 11.3:

```
prot <- sapply(splits,`[`,1)
prot
```

```
[1] "PROT300" "PROT10"  "PROT5"
```

Things would be more complicated if strings had more than one dash, but the example here illustrates the principle. We say `fixed=TRUE` above because the `split` argument is otherwise interpreted as a regular expression, giving special meaning to dots and slashes, etc.

11.3 Stratified Imputation

Recall that in Chapter 6, we tried to perform a vertical imputation of weight, where one of our subjects was missing the first weight observation (Listing 6.27). `locf()` won't work here, because Subject `b` will have a weight imputed using data from subject `a`.

Listing 11.4:

```
pk <- data.frame(
       subject=rep(letters[1:3],each=3),
       time=rep(1:3,3),
       weight=c(70,70.5,69,NA,71,70,72, 72,NA)
)
pk
```

```
  subject time weight
1       a    1   70.0
2       a    2   70.5
3       a    3   69.0
4       b    1     NA
5       b    2   71.0
6       b    3   70.0
7       c    1   72.0
8       c    2   72.0
9       c    3     NA
```

What we need here is a stratified imputation. That is, we need to conduct the imputation independently for subsets of the data. In Listing 6.33, we used `reapply` for aggregation. `reapply` can also be used for stratified imputation, because it always returns a vector as long as its primary input.

Listing 11.5:

```
library(metrumrg)
pk$WEIGHT <- reapply(pk$weight,INDEX=pk$subject,FUN=locf)
pk
```

```
  subject time weight WEIGHT
1       a    1   70.0   70.0
2       a    2   70.5   70.5
3       a    3   69.0   69.0
4       b    1     NA     NA
5       b    2   71.0   71.0
6       b    3   70.0   70.0
7       c    1   72.0   72.0
8       c    2   72.0   72.0
9       c    3     NA   72.0
```

Note that the first value for subject b is still missing, as it should be. The last value for subject c has been imputed correctly.

11.4 Dynamic Transformations

Consider these data tables (source is in the Example Code section at the end of the chapter).

Listing 11.6:

```
dose
```

```
  subject time amount
1       a    0     40
2       a    2     40
3       b    1     60
4       b    3     60
5       c    0     80
6       c    2     80
```

Listing 11.7:

```
pk
```

```
   subject time conc pain
1        a    0 3.30    5
2        a    1 1.70   NA
3        a    2 3.30    4
4        a    3 3.30    3
5        b    0 2.40   NA
6        b    1 0.46    6
7        b    2 1.10    4
8        b    3 1.70    3
9        c    0 2.00    4
10       c    1 4.40    5
11       c    2 2.80    3
12       c    3 1.20   NA
```

Suppose we wish to analyze a compartmental model with half the dose going into compartment 1 and half to compartment 2. We could make two copies our dose data, rescale the amounts, and add a compartment variable to each. Then of course we would merge them with our pk data. However, if we discovered later that the dose data needed changes, we would have to be sure to make the same changes to each copy. And if we want to consider a traditional model simultaneously, we may need *three* copies of the dose data, one like the original.

A better way to approach the problem is to leave the source data untouched (it's already as 'correct' as we can make it) but change the way it is used. We'll call this **dynamic transformation**. The compartmental datasets we suggested earlier will never actually exist as stored

objects. We'll create them on the fly, as needed. The fewer 'versions' of the data that are lying around, the better.

The `transform()` function lets us create variables dynamically (their names are the names of the supplied arguments). Values can be specified in terms of columns present in the original table.

Listing 11.8:

```
merge(
  merge(
    transform(dose, amount=amount/2,compartment=1),
    transform(dose, amount=amount/2,compartment=2),
    all=TRUE
  ),
  pk,
  all=TRUE
)
```

```
   subject time amount compartment conc pain
1        a    0     20           1 3.30    5
2        a    0     20           2 3.30    5
3        a    1     NA          NA 1.70   NA
4        a    2     20           1 3.30    4
5        a    2     20           2 3.30    4
6        a    3     NA          NA 3.30    3
7        b    0     NA          NA 2.40   NA
8        b    1     30           1 0.46    6
9        b    1     30           2 0.46    6
10       b    2     NA          NA 1.10    4
11       b    3     30           1 1.70    3
12       b    3     30           2 1.70    3
13       c    0     40           1 2.00    4
14       c    0     40           2 2.00    4
15       c    1     NA          NA 4.40    5
16       c    2     40           1 2.80    3
17       c    2     40           2 2.80    3
18       c    3     NA          NA 1.20   NA
```

Listing 11.9:

We may also want to specify the compartment for the pk values, and think carefully about the row order for pk and dose records occuring at the same time.

11.5 Complex Criteria

Consider this data set, which uses Example Code data.

Listing 11.10:

```
data <- merge(
  transform(dose,evid=1),
  transform(pk,evid=0),
  all=TRUE
)
data
```

```
   subject time evid amount conc pain
1        a    0    0     NA 3.30    5
2        a    0    1     40   NA   NA
3        a    1    0     NA 1.70   NA
4        a    2    0     NA 3.30    4
5        a    2    1     40   NA   NA
6        a    3    0     NA 3.30    3
7        b    0    0     NA 2.40   NA
8        b    1    0     NA 0.46    6
9        b    1    1     60   NA   NA
10       b    2    0     NA 1.10    4
11       b    3    0     NA 1.70    3
12       b    3    1     60   NA   NA
13       c    0    0     NA 2.00    4
14       c    0    1     80   NA   NA
15       c    1    0     NA 4.40    5
16       c    2    0     NA 2.80    3
17       c    2    1     80   NA   NA
18       c    3    0     NA 1.20   NA
```

Suppose we wish to ignore all non-zero concentrations that occur before the first dose. How can we identify the ignorable rows? The criteria are complex: `evid` should be zero, and `conc` should be non-zero: horizontal criteria. But `evid 0` should also appear before all `evid 1`: a vertical criterion. Additionally, the evaluation should be stratified: that is, within subject.

The problem becomes simpler when we realize that, in the end, we will have made one evaluation per row. Essentially, then, we will have to reduce the 'vertical' elements of the problem to 'horizontal' elements, whether implicitly or explicitly.

To make this clearer, let's solve the problem first for just one subject.

Listing 11.11:

```
b <- data[data$subject=='b',]
```

The vertical comparison involves 'before', which is generally a question about time, and specifically a question about record order. If we can calculate a 'before' value for each record, the problem becomes strictly horizontal.

Listing 11.12:

```
b$before1stDose <- before(b$evid==1)
b
```

```
   subject time evid amount conc pain before1stDose
7        b    0    0     NA 2.40   NA           TRUE
8        b    1    0     NA 0.46    6           TRUE
9        b    1    1     60   NA   NA          FALSE
10       b    2    0     NA 1.10    4          FALSE
11       b    3    0     NA 1.70    3          FALSE
12       b    3    1     60   NA   NA          FALSE
```

Now we can do the horizontal evaluations (one per row) all at once.

Listing 11.13:

```
b$ignore <- with(
  b,
  evid==0 &
  !is.na(conc)
  & conc > 0
  & before1stDose
)
b
```

```
   subject time evid amount conc pain before1stDose ignore
7        b    0    0     NA 2.40   NA           TRUE   TRUE
8        b    1    0     NA 0.46    6           TRUE   TRUE
9        b    1    1     60   NA   NA          FALSE  FALSE
10       b    2    0     NA 1.10    4          FALSE  FALSE
11       b    3    0     NA 1.70    3          FALSE  FALSE
12       b    3    1     60   NA   NA          FALSE  FALSE
```

The same approach works when solving simultaneously for all subjects. We specify `within` to nest the calculation within subject.

Listing 11.14:

```
data$before1stDose <- with(
  data,
```

```
  before(
    evid==1,
    within=subject
  )
)
data$ignore <- with(data, evid==0 & !is.na(conc) & conc > 0 &
   before1stDose)
data
```

```
   subject time evid amount conc pain before1stDose ignore
1        a    0    0     NA 3.30    5          TRUE   TRUE
2        a    0    1     40   NA   NA         FALSE  FALSE
3        a    1    0     NA 1.70   NA         FALSE  FALSE
4        a    2    0     NA 3.30    4         FALSE  FALSE
5        a    2    1     40   NA   NA         FALSE  FALSE
6        a    3    0     NA 3.30    3         FALSE  FALSE
7        b    0    0     NA 2.40   NA          TRUE   TRUE
8        b    1    0     NA 0.46    6          TRUE   TRUE
9        b    1    1     60   NA   NA         FALSE  FALSE
10       b    2    0     NA 1.10    4         FALSE  FALSE
11       b    3    0     NA 1.70    3         FALSE  FALSE
12       b    3    1     60   NA   NA         FALSE  FALSE
13       c    0    0     NA 2.00    4          TRUE   TRUE
14       c    0    1     80   NA   NA         FALSE  FALSE
15       c    1    0     NA 4.40    5         FALSE  FALSE
16       c    2    0     NA 2.80    3         FALSE  FALSE
17       c    2    1     80   NA   NA         FALSE  FALSE
18       c    3    0     NA 1.20   NA         FALSE  FALSE
```

For this particular data set, `ignore` happens to be identical to `before1stDose`, but that may not be true in other cases.

11.6 Table Reorganization

In section 6.4.2, we said that the `key` columns of a table should identify unique objects and the other columns should list attributes of those objects. More generally, we could say that a table is just a collection of values that are distinguished either by row criteria or column criteria. The values here are the object attributes, and the row criteria are the keys.

The attribute-centric view of table structure raises an interesting possibility. If values can be distinguished either by rows or by columns, perhaps we can interconvert *row-level* distinctions and *column-level* distinctions. The following two tables give the same information.

```
  subject  side injections
1       a  left          5
2       a right          6
3       b  left          7
4       b right          8

  subject left right
1       a    5     6
2       b    7     8
```

In the first table, all the distinctions among injection counts are made with row-level combinations of values, e.g. 'a-left'. In the second table, the counts are distinguished partly by rows, and partly by columns.

The ability to convert easily between row-level and column-level distinctions is of great value. The package `reshape` by Hadley Wickham gives us a general strategy for doing just that. Wickham's technique is to create an intermediate table where all distinctions are row-level distinctions, using the function `melt()`. This 'molten' table can be 'cast' into any desired combination of row-level and column-level distinctions, using the function `cast()`.

11.6.1 melt

Consider this blood pressure data.

Listing 11.15:

```
a <- data.frame(
  subject=c('a','a','b','b'),
  position=c('sit','stand','sit','stand'),
  SBP=c(120,130,115,125),
  DBP=c(80,90,75,85)
)
a
```

```
  subject position SBP DBP
1       a      sit 120  80
2       a    stand 130  90
3       b      sit 115  75
4       b    stand 125  85
```

We can convert everything to a row-level distinction with `melt()`. We identify the key columns with the argument `id.var`, and we identify the values columns with the argument `measure.var`.

Listing 11.16:

```
molten <- melt(
  a,
  id.var=c('subject','position'),
  measure.var=c('SBP','DBP')
)
molten
```

```
  subject position variable value
1       a      sit      SBP   120
2       a    stand      SBP   130
3       b      sit      SBP   115
4       b    stand      SBP   125
5       a      sit      DBP    80
6       a    stand      DBP    90
7       b      sit      DBP    75
8       b    stand      DBP    85
```

11.6.2 cast

In the original data, we knew '120' was systolic because of a column-level distinction. We knew it was a sitting blood pressure because of a row-level distinction. Let's swap the two.

The function `cast()` expects a molten data frame and a formula. The formula has the form `X~Y`, where `X` indicates the row-level distinctions and `Y` indicates the column-level distinctions. Here, we use rows to distinguish subject and type, while using columns to distinguish position.

Listing 11.17:

```
cast(molten, subject + variable  ~  position)
```

```
  subject variable sit stand
1       a      SBP 120   130
2       a      DBP  80    90
3       b      SBP 115   125
4       b      DBP  75    85
```

We could easily get the original form back as well.

Listing 11.18:

```
cast(molten, subject + position  ~  variable)
```

```
  subject position SBP DBP
1       a      sit 120  80
2       a    stand 130  90
3       b      sit 115  75
4       b    stand 125  85
```

The functions `melt()` and `cast()` are very powerful. Proper use of the formula requires practice, but effort here will be rewarded. Reshaping a data set can simplify plotting tasks; frequently, a molten data.frame is just what you need as an argument to `xyplot()`. Also, `cast()` gives a powerful alternative for aggregating data.frames. If your formula does not completely distinguish all values from each other, groups of values with matching distinctions will be aggregated as specified by the argument `fun.aggregate()`. See the help for details.

11.7 Exercises

1. For the `id` variable in Listing 11.1, extract just the subject numbers.
2. For `pk` in the Example Code below, impute missing pain scores using `locf()`.
3. Merge `dose` and `pk` in the Example Code, dynamically creating a `type` variable that is 'dose' or 'pk', respectively.
4. For `data` in Listing 11.10 (using `dose` and `pk` from the Example Code), create an `ignore` flag that is `TRUE` for positive `conc` values that occur before the first dose, *except* where `pain` is not `NA`.
5. For `data` in Listing 11.10, create an `ignore` flag for doses that occur after the last sample.
6. Convert the data in Listing 11.15 so that 'subject' is a column-level distinction and sit/stand and SBP/DBP are row-level distinctions.
7. For `data` in Listing 11.10, combine `conc` and `pain` into a single column. Sort the result on subject, time, and evid. Then create an `ignore` flag for `conc` values that occur before the first dose.
8. For `pk` from the Example Code, drop the `pain` scores and arrange the `conc` values using a separate column for each time.

11.8 Example Code

Listing 11.19:

```
set.seed(0)
dose <- data.frame(
       subject = rep(letters[1:3], each = 2),
       time = c(0,2,1,3,0,2),
       amount = rep(c(40,60,80), each = 2)
)
pk <- data.frame(
       subject = rep(letters[1:3], each = 4),
       time = c(0,1,2,3,0,1,2,3,0,1,2,3),
       conc = signif(rnorm(12)+2,2),
        pain =c(5,NA,4,3,NA,6,4,3,4,5,3,NA)
)
demo <- data.frame(
       subject = letters[1:4],
       race = c('asian','white','black','other'),
       sex = c('female','male','female','male'),
       weight = c(75, 70, 73, 68)
)
```

Chapter 12

Advanced Graphics

12.1 Objectives

After completing this chapter, you will . . .

- understand the fundamentals of grid units, including "null", "npc", and "native" units
- understand the fundamentals of navigating grid viewports
- understand the fundamentals of grid layouts

12.2 Introduction

As was discussed in Chapter 2, much of the important graphics functionality in R builds upon the *lattice* package or builds directly upon the *grid* graphics package. In previous chapters we examined the basic functionality provided by the *lattice* package; we now turn to the *grid* package.

We begin with some truth in advertising: for day-to-day applied pharmacometric work, it may *not* be worth the effort to construct customized displays "from scratch" using *grid*. In general, if you can find a way to get the job done with *lattice*, you will save yourself a lot of time. Having said that, knowledge of *grid* will help you better understand how *lattice* works, so that you can augment and make changes to basic *lattice* functionality.

A comprehensive treatment of *grid* graphics is provided in the book *R Graphics* by Paul Murrell (the author of the *grid* package) [1].

12.3 Grid Basics

We begin by loading the *grid* package:

Listing 12.1:

```
library(grid)
```

To create a blank slate that we can draw on with grid commands, we can do:

Listing 12.2:

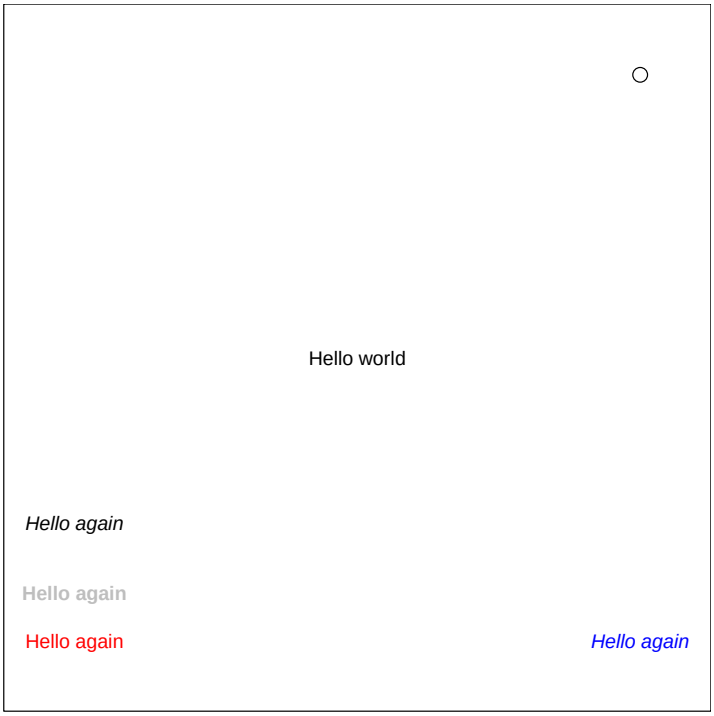
```
grid.newpage()
```

12.3.1 Units

We can draw right in the `ROOT` viewport using grid primitives such as `grid.text()` and `grid.points()`:

Listing 12.3:

```
grid.newpage()
grid.text("Hello world")
grid.text("Hello again", 0.1, 0.1, gp = gpar(col = "red"))
grid.text("Hello again", 0.9, 0.1, gp = gpar(col = "blue",
   fontface = "italic"))
grid.text("Hello again", 0.1, unit(1, "inches"), gp = gpar(col = "
   grey", fontface = "bold"))
grid.text("Hello again", 0.1, unit(0.1, "npc") + unit(1, "inches")
   , gp = gpar(fontface = "oblique"))
grid.points(0.9, 0.9, default.units = "npc") # normalized parent
   coordinates
grid.rect(gp = gpar(fill='transparent'))
```



In our first call to `grid.text()`, the default positioning was used, putting the text right in the middle of the plot (technically, right in the middle of the current viewport - see next section). In our next two calls to `grid.text()`, we specified both coordinates on a 0–1 scale. In *grid* terminology, 0–1 coordinates are referred to as *normalized parent coordinates (npc)*. The function `grid.text` uses *npc* units by default, so we didn't need to specifically request this. In our fourth call to `grid.text()`, we specified the *x* coordinate in terms of *npc* units and the *y* coordinate in terms of inches. This easy mixing and matching of different units is part of the power of *grid*. We can even do basic artithmetic on different types of units: see the fifth call to `grid.text()`.

The function `grid.points()` uses a different default type of unit to interpret coordinates, however we were still able to use *npc* units by explicitly specifying `default.units = ``npc''` The default units used by `grid.points()` are called *native* units. In order to use *native* units, we first need to create a new *viewport*.

12.3.2 Viewports

All *grid* drawing takes place inside of "viewports". Each viewport has its own coordinate system, and *grid* functionality allows one to navigate between these different coordinate systems. When we first start, we are in the `ROOT` viewport.

Listing 12.4:

```
grid.newpage()
```

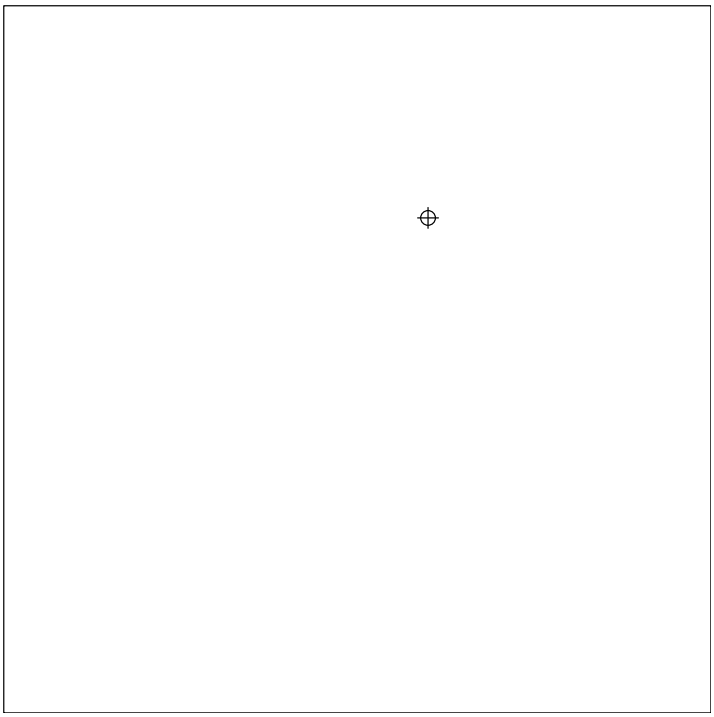
```
current.viewport()
```

```
viewport[ROOT]
```

The `ROOT` viewport does not have any *native* units associated with it. *Native* units are a type of units defined by the user-defined scale of a plotting region. Here we "push on" on a viewport with *native* units ranging from 0 to 10 on both axes, and we draw two points in the same place in order to show the correspondance between the *npc* and *native* coordinates.

Listing 12.5:

```
grid.newpage()
pushViewport(viewport(xscale = c(0, 10), yscale = c(0, 10)))
grid.points(6, 7, pch = 1) # circle plot character
grid.points(0.6, 0.7, default.units = "npc", pch = 3) # cross plot
    character
grid.rect(gp=gpar(fill='transparent'))
```

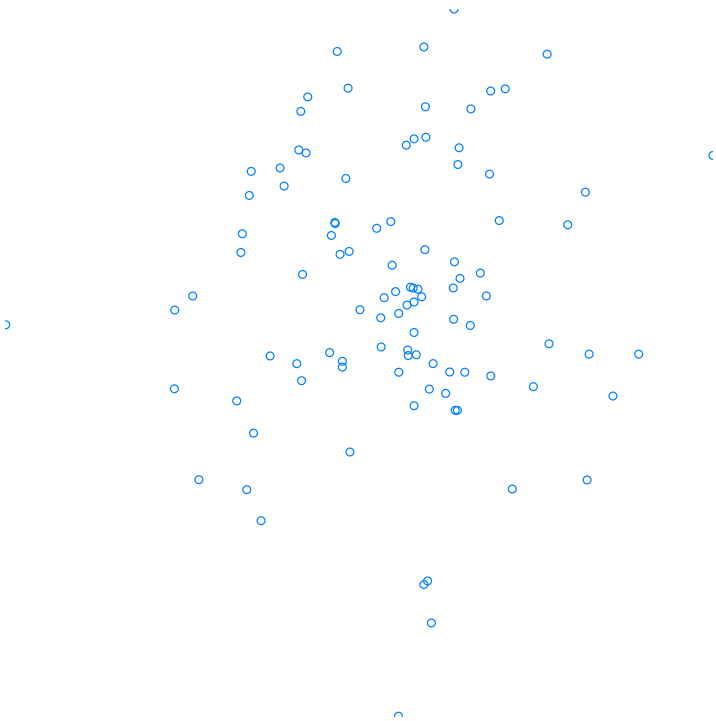


The *lattice* panel functions that we used in previous chapters are designed to do most of their drawing using *native* coordinates. We can take advantage of this to use panel functions outside the context of a complete *lattice* display:

Listing 12.6:

```
grid.newpage()
fakedata <- data.frame(X = rnorm(100), Y = rnorm(100))
```

```
pushViewport(viewport(xscale = range(fakedata$X),
                      yscale = range(fakedata$Y)
                      )
             )
library(lattice)
panel.xyplot(fakedata$X, fakedata$Y)
```



One helpful function to know about for this kind of work is `extend.limits` from the *lattice* package. This function is not generally exposed to the user, but we can fish it out from the *lattice* namespace as follows:

Listing 12.7:

```
extend.limits <- lattice:::extend.limits
```

The mundane but important job of `extend.limits()` is to add a bit of padding to around the extremes of the data:

Listing 12.8:

```
range(fakedata$X)
```

```
[1] -3.233152  2.919140
```

Listing 12.9:

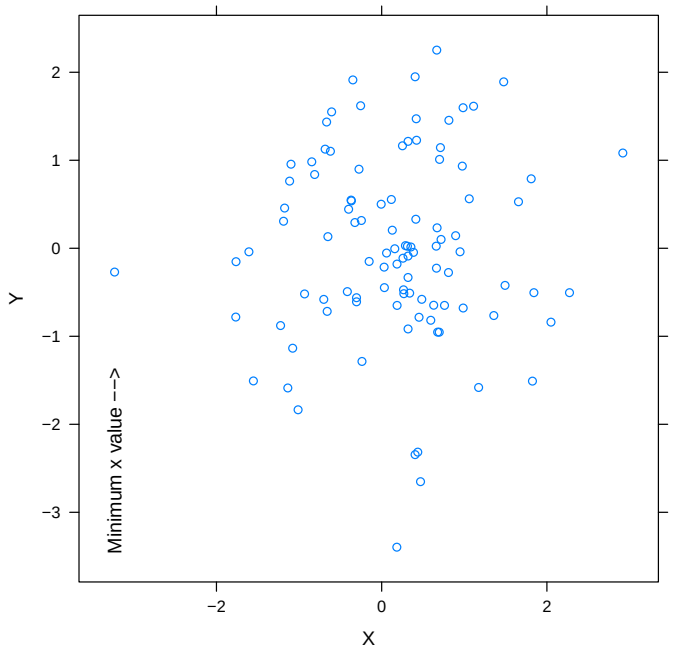
```
extend.limits(range(fakedata$X))
```

```
[1] -3.663813  3.349801
```

The example above shows how to use lattice functionality within a general *grid* graphic. We can also do the reverse: leverage *grid* functions when making our *lattice* plots:

Listing 12.10:

```
print(xyplot(Y  ~  X, data = fakedata,
             panel = function(x, y, ...) {
               panel.xyplot(x, y, ...)
               grid.text("Minimum x value -->",
                         unit(min(x), "native"),
                         unit(0.05, "npc"),
                         just = "left",
                         rot = 90
                         )
             }
             )
      )
```



12.4 Layouts

We now turn to the concept of the *grid* layout. In the following example, we divide the total graphics area into a 3×3 layout, and position our scatterplot in the second column and first row of this layout:

Listing 12.11:

```
grid.newpage()
lyt <- grid.layout(3, 3)
pushViewport(viewport(layout = lyt))
current.vpTree()
```

```
viewport[ROOT]->(viewport[GRID.VP.5])
```

Listing 12.12:

```
xlim <- extend.limits(range(fakedata$X))
ylim <- extend.limits(range(fakedata$Y))
pushViewport(viewport(layout.pos.row = 2, layout.pos.col = 1,
                      xscale = xlim,
                      yscale = ylim
                      )
             )
current.vpTree()
```

```
viewport[ROOT]->(viewport[GRID.VP.5]->(viewport[GRID.VP.6]))
```

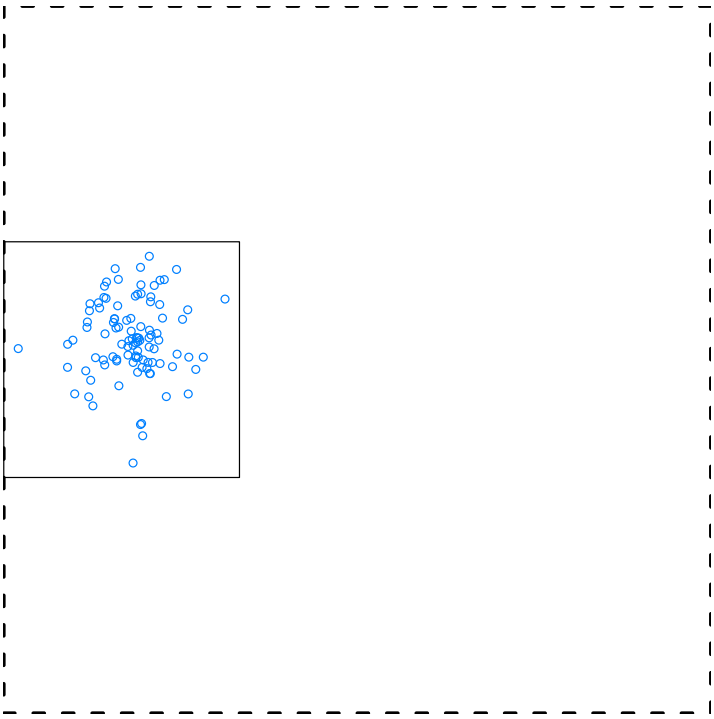
Listing 12.13:

```
grid.rect(gp = gpar(fill='transparent'))
panel.xyplot(fakedata$X, fakedata$Y)
popViewport()
current.vpTree()
```

```
viewport[ROOT]->(viewport[GRID.VP.5])
```

Listing 12.14:

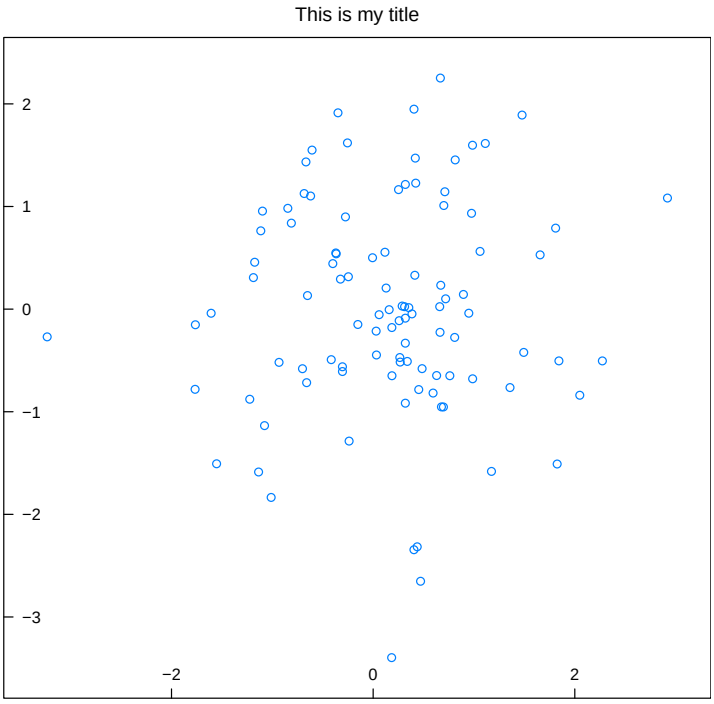
```
grid.rect(gp = gpar(fill='transparent', lwd = 3, lty = 2))
```



In the previous example, all elements of our layout were sized relative to the overall size of the plotting area (and will scale with the overall size of the figure). Sometimes we instead want to reserve an absolute amount of space for some parts of the graphic. In the following example, we reserve a fixed height of 1 cm for the height of the title, and allow the plot to occupy whatever space is left (this is the meaning of 1 "null" unit).

Listing 12.15:

```
grid.newpage()
lyt2 <- grid.layout(2, 1, heights = unit.c(unit(1, "cm"), unit(1,
   "null")))
pushViewport(viewport(layout = lyt2))
pushViewport(viewport(layout.pos.row = 1))
grid.text("This is my title")
popViewport()
pushViewport(viewport(layout.pos.row = 2, xscale = xlim, yscale =
   ylim))
panel.xyplot(fakedata$X, fakedata$Y)
panel.axis("bottom", half = FALSE)
panel.axis("left", half = FALSE)
grid.rect(gp=gpar(fill='transparent'))
```



12.5 A Practical Example

The *HH* package implements a number of graphical representations for adverse event data, as proposed in a recent paper by Heiberger and Holland [5]. As an illustrative exercise, we are going to modify a function called `ae.dotplot()`, found in this package. This function makes a plot that is generally useful for situations where proportions are being compared.

We take some lines from the help file for `ae.dotplot()` to construct an example data set:

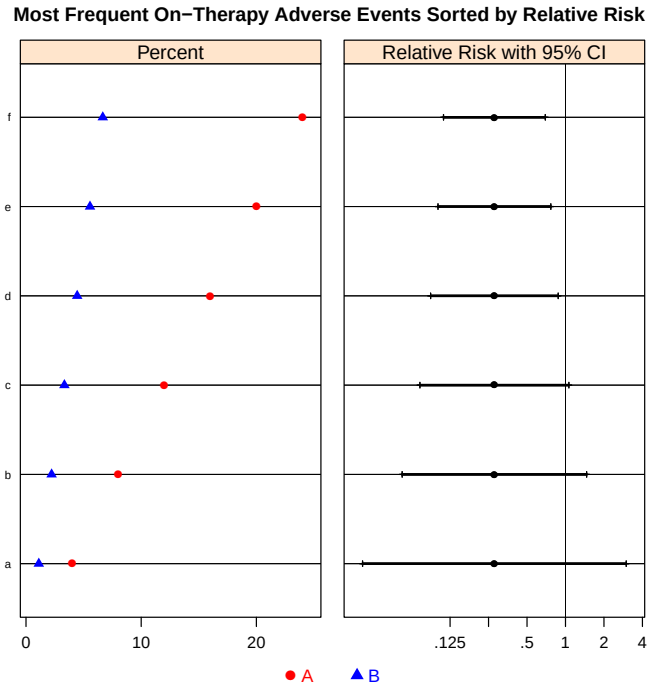
Listing 12.16:

```
library(HH)
# From the HH documentation:
aewide <- data.frame(Event=letters[1:6],
                     N.A=c(50,50,50,50,50,50),
                     N.B=c(90,90,90,90,90,90),
                     AE.A=2*(1:6),
                     AE.B=1:6)
aewtol <- aeReshapeToLong(aewide)
xr <- logrelrisk(aewtol)
```

Here is what you get when you use `ae.dotplot()` "out of the box":

Listing 12.17:

```
print(
      ae.dotplot(xr)
      )
```

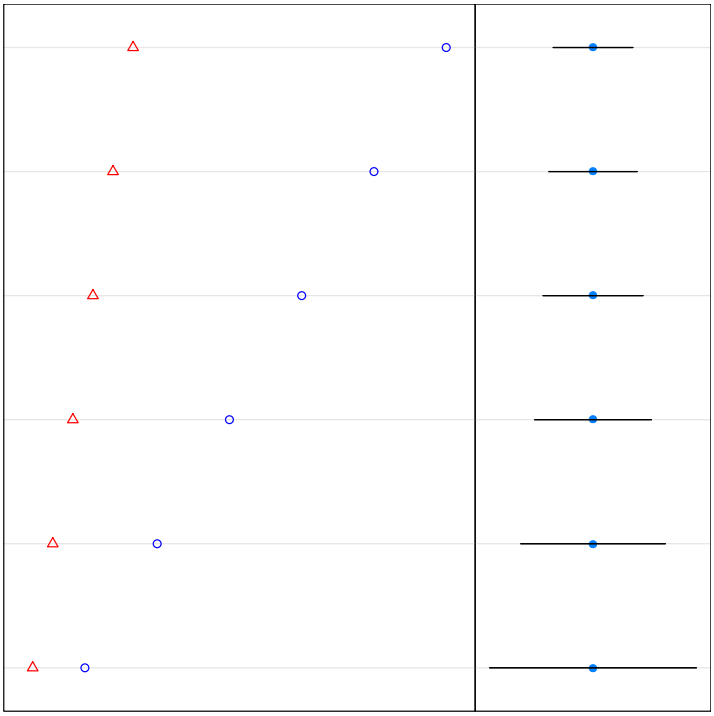


This "out-of-the-box" behavior is likely to be sufficient for most purposes. However, as an exercise let's suppose that we want the panel on the right to be only half as wide as the panel on the left.

Listing 12.18:

```
grid.newpage()
pctlim <- extend.limits(range(xr$PCT))
lrrlim <- extend.limits(range(xr[c("logrelriskCI.lower", "
   logrelriskCI.upper")]))
preflim <- extend.limits(c(1, length(unique(xr$PREF))))
pushViewport(viewport(layout = grid.layout(1, 2,
                        widths = c(2, 1)
                        )
                      )
             )
pushViewport(viewport(layout.pos.col = 1, xscale = pctlim, yscale
   = preflim))
```

```
panel.dotplot(xr$PCT, xr$PREF,
                  pch = (1:2)[as.factor(xr$RAND)],
                  col = c("blue", "red")[as.factor(xr$RAND)]
                  )
grid.rect(gp =gpar(fill='transparent'))
popViewport()
pushViewport(viewport(layout.pos.col = 2, xscale = lrrlim, yscale
   = preflim))
panel.dotplot(xr$logrelrisk, xr$PREF)
panel.segments(xr$logrelriskCI.lower, xr$PREF, xr$logrelriskCI.
   upper, xr$PREF)
grid.rect(gp = gpar(fill='transparent'))
```



We are still a long way from having a production quality plot. To finish the job, we would need to make space for tick marks, tick mark labels, axis labels, a legend, and a title (one comes to appreciate the burden that is usually carried by `xyplot()` and other high level plotting functions). These remaining steps, while tedious, would not require any knowledge beyond what we have already discussed.

Appendix A

Complete Courseware License

Creative Commons Legal Code

Attribution-NonCommercial-ShareAlike 3.0 Unported

CREATIVE COMMONS CORPORATION IS NOT A LAW FIRM AND DOES NOT PROVIDE LEGAL SERVICES. DISTRIBUTION OF THIS LICENSE DOES NOT CREATE AN ATTORNEY-CLIENT RELATIONSHIP. CREATIVE COMMONS PROVIDES THIS INFORMATION ON AN "AS-IS" BASIS. CREATIVE COMMONS MAKES NO WARRANTIES REGARDING THE INFORMATION PROVIDED, AND DISCLAIMS LIABILITY FOR DAMAGES RESULTING FROM ITS USE.

License

THE WORK (AS DEFINED BELOW) IS PROVIDED UNDER THE TERMS OF THIS CREATIVE COMMONS PUBLIC LICENSE ("CCPL" OR "LICENSE"). THE WORK IS PROTECTED BY COPYRIGHT AND/OR OTHER APPLICABLE LAW. ANY USE OF THE WORK OTHER THAN AS AUTHORIZED UNDER THIS LICENSE OR COPYRIGHT LAW IS PROHIBITED.

BY EXERCISING ANY RIGHTS TO THE WORK PROVIDED HERE, YOU ACCEPT AND AGREE TO BE BOUND BY THE TERMS OF THIS LICENSE. TO THE EXTENT THIS LICENSE MAY BE CONSIDERED TO BE A CONTRACT, THE LICENSOR GRANTS YOU THE RIGHTS CONTAINED HERE IN CONSIDERATION OF YOUR ACCEPTANCE OF SUCH TERMS AND CONDITIONS.

1. Definitions

a. **"Adaptation"** means a work based upon the Work, or upon the Work and other pre-existing works, such as a translation, adaptation, derivative work, arrangement of music or other alterations of a literary or artistic work, or phonogram or performance and includes cinematographic adaptations or any other form in which the Work may be recast, transformed, or adapted including in any form recognizably derived from the original, except that a work that constitutes a Collection will not be considered an Adaptation for the purpose of this License. For the avoidance of doubt, where the Work is a musical work, performance or phonogram, the synchronization of the Work in timed-relation with a moving image ("synching") will be considered an Adaptation for the purpose of this License.
b. **"Collection"** means a collection of literary or artistic works, such as encyclopedias and anthologies, or performances, phonograms or broadcasts, or other works or subject matter other than works listed in Section 1(g) below, which, by reason of the selection and arrangement of their contents, constitute intellectual creations, in which the Work is included in its entirety in unmodified form along with one or more other contributions, each constituting separate and independent works in themselves, which together are assembled into a collective whole. A work that constitutes a Collection will not be considered an Adaptation (as defined above) for the purposes of this License.
c. **"Distribute"** means to make available to the public the original and copies of the Work or Adaptation, as appropriate, through sale or other transfer of ownership.
d. **"License Elements"** means the following high-level license attributes as selected by Licensor and indicated in the title of this License: Attribution, Noncommercial, ShareAlike.
e. **"Licensor"** means the individual, individuals, entity or entities that offer(s) the Work under the terms of this License.
f. **"Original Author"** means, in the case of a literary or artistic work, the individual, individuals, entity or entities who created the Work or if no individual or entity can be identified, the publisher; and in

addition (i) in the case of a performance the actors, singers, musicians, dancers, and other persons who act, sing, deliver, declaim, play in, interpret or otherwise perform literary or artistic works or expressions of folklore; (ii) in the case of a phonogram the producer being the person or legal entity who first fixes the sounds of a performance or other sounds; and, (iii) in the case of broadcasts, the organization that transmits the broadcast.

g. **"Work"** means the literary and/or artistic work offered under the terms of this License including without limitation any production in the literary, scientific and artistic domain, whatever may be the mode or form of its expression including digital form, such as a book, pamphlet and other writing; a lecture, address, sermon or other work of the same nature; a dramatic or dramatico-musical work; a choreographic work or entertainment in dumb show; a musical composition with or without words; a cinematographic work to which are assimilated works expressed by a process analogous to cinematography; a work of drawing, painting, architecture, sculpture, engraving or lithography; a photographic work to which are assimilated works expressed by a process analogous to photography; a work of applied art; an illustration, map, plan, sketch or three-dimensional work relative to geography, topography, architecture or science; a performance; a broadcast; a phonogram; a compilation of data to the extent it is protected as a copyrightable work; or a work performed by a variety or circus performer to the extent it is not otherwise considered a literary or artistic work.
h. **"You"** means an individual or entity exercising rights under this License who has not previously violated the terms of this License with respect to the Work, or who has received express permission from the Licensor to exercise rights under this License despite a previous violation.
i. **"Publicly Perform"** means to perform public recitations of the Work and to communicate to the public those public recitations, by any means or process, including by wire or wireless means or public digital performances; to make available to the public Works in such a way that members of the public may access these Works from a place and at a place individually chosen by them; to perform the Work to the public by any means or process and the communication to the public of the performances of the Work, including by public digital performance; to broadcast and rebroadcast the Work by any means including signs, sounds or images.
j. **"Reproduce"** means to make copies of the Work by any means including without limitation by sound or visual recordings and the right of fixation and reproducing fixations of the Work, including storage of a protected performance or phonogram in digital form or other electronic medium.

2. Fair Dealing Rights. Nothing in this License is intended to reduce, limit, or restrict any uses free from copyright or rights arising from limitations or exceptions that are provided for in connection with the copyright protection under copyright law or other applicable laws.

3. License Grant. Subject to the terms and conditions of this License, Licensor hereby grants You a worldwide, royalty-free, non-exclusive, perpetual (for the duration of the applicable copyright) license to exercise the rights in the Work as stated below:

a. to Reproduce the Work, to incorporate the Work into one or more Collections, and to Reproduce the Work as incorporated in the Collections;
b. to create and Reproduce Adaptations provided that any such Adaptation, including any translation in any medium, takes reasonable steps to clearly label, demarcate or otherwise identify that changes were made to the original Work. For example, a translation could be marked "The original work was translated from English to Spanish," or a modification could indicate "The original work has been modified.";
c. to Distribute and Publicly Perform the Work including as incorporated in Collections; and,
d. to Distribute and Publicly Perform Adaptations.

The above rights may be exercised in all media and formats whether now known or hereafter devised. The above rights include the right to make such modifications as are technically necessary to exercise the rights in other media and formats. Subject to Section 8(f), all rights not expressly granted by Licensor are hereby reserved, including but not limited to the rights described in Section 4(e).

4. Restrictions. The license granted in Section 3 above is expressly made subject to and limited by the following restrictions:

a. You may Distribute or Publicly Perform the Work only under the terms of this License. You must include a copy of, or the Uniform Resource Identifier (URI) for, this License with every copy of the Work You Distribute or Publicly Perform. You may not offer or impose any terms on the Work that restrict the terms of this License or the ability of the recipient of the Work to exercise the rights granted to that recipient under the terms of the License. You may not sublicense the Work. You must keep intact all notices that refer to this License and to the disclaimer of warranties with every copy of the Work You Distribute or Publicly Perform. When You Distribute or Publicly Perform the Work, You may not impose any effective technological measures on the Work that restrict the ability of a recipient of the Work from You to exercise the rights granted to that recipient under the terms of the License. This Section 4(a) applies to the Work as incorporated in a Collection, but this does not require the Collection apart from the Work itself to be made subject to the terms of this License. If You create a Collection, upon notice from any Licensor You must, to the extent practicable, remove from the Collection any credit as required by Section 4(d), as requested. If You create an Adaptation, upon notice from any Licensor You must, to the extent practicable, remove from the Adaptation any credit as required by Section 4(d), as requested.
b. You may Distribute or Publicly Perform an Adaptation only under: (i) the terms of this License; (ii) a later version of this License with the same License Elements as this License; (iii) a Creative Commons jurisdiction license (either this or a later license version) that contains the same License Elements as this License (e.g., Attribution-NonCommercial-ShareAlike 3.0 US) ("Applicable License"). You must include a copy of, or the URI, for Applicable License with every copy of each Adaptation You Distribute or Publicly Perform. You may not offer or impose any terms on the Adaptation that restrict the terms of the Applicable License or the ability of the recipient of the Adaptation to exercise the rights granted to that recipient under the terms of the Applicable License. You must keep intact all notices that refer to the Applicable License and to the disclaimer of warranties with every copy of the Work as included in the Adaptation You Distribute or Publicly Perform. When You Distribute or Publicly Perform the Adaptation, You may not impose any effective technological measures on the Adaptation that restrict the ability of a recipient of the Adaptation from You to exercise the rights granted to that recipient under the terms of the Applicable License. This Section 4(b) applies to the Adaptation as incorporated in a Collection, but this does not require the Collection apart from the Adaptation itself to be made subject to the terms of the Applicable License.
c. You may not exercise any of the rights granted to You in Section 3 above in any manner that is primarily intended for or directed toward commercial advantage or private monetary compensation. The exchange of the Work for other copyrighted works by means of digital file-sharing or otherwise shall not be considered to be intended for or directed toward commercial advantage or private monetary compensation, provided there is no payment of any monetary compensation in connection with the exchange of copyrighted works.
d. If You Distribute, or Publicly Perform the Work or any Adaptations or Collections, You must, unless a request has been made pursuant to Section 4(a), keep intact all copyright notices for the Work and provide, reasonable to the medium or means You are utilizing: (i) the name of the Original Author (or pseudonym, if applicable) if supplied, and/or if the Original Author and/or Licensor designate another party or parties (e.g., a sponsor institute, publishing entity, journal) for attribution ("Attribution Parties") in Licensor's copyright notice, terms of service or by other reasonable means, the name of such party or parties; (ii) the title of the Work if supplied; (iii) to the extent reasonably practicable, the URI, if any, that Licensor specifies to be associated with the Work, unless such URI does not refer to the copyright notice or licensing information for the Work; and, (iv) consistent with Section 3(b), in the case of an Adaptation, a credit identifying the use of the Work in the Adaptation (e.g., "French translation of the Work by Original Author," or "Screenplay based on original Work by Original Author"). The credit required by this Section 4(d) may be implemented in any reasonable manner; provided, however, that in the case of a Adaptation or Collection, at a minimum such credit will appear, if a credit for all contributing authors of the Adaptation or Collection appears, then as part of these credits and in a manner at least as prominent as the credits for the other contributing authors. For the avoidance of doubt, You may only use the credit required by this Section for the purpose of attribution in the manner set out above and, by exercising Your rights under this License, You may not implicitly or explicitly assert or imply any connection with, sponsorship or endorsement by the Original Author, Licensor and/or Attribution Parties, as appropriate, of You or Your use of the Work, without the separate, express prior written

permission of the Original Author, Licensor and/or Attribution Parties.

e. For the avoidance of doubt:

   i. **Non-waivable Compulsory License Schemes**. In those jurisdictions in which the right to collect royalties through any statutory or compulsory licensing scheme cannot be waived, the Licensor reserves the exclusive right to collect such royalties for any exercise by You of the rights granted under this License;
   ii. **Waivable Compulsory License Schemes**. In those jurisdictions in which the right to collect royalties through any statutory or compulsory licensing scheme can be waived, the Licensor reserves the exclusive right to collect such royalties for any exercise by You of the rights granted under this License if Your exercise of such rights is for a purpose or use which is otherwise than noncommercial as permitted under Section 4(c) and otherwise waives the right to collect royalties through any statutory or compulsory licensing scheme; and,
   iii. **Voluntary License Schemes**. The Licensor reserves the right to collect royalties, whether individually or, in the event that the Licensor is a member of a collecting society that administers voluntary licensing schemes, via that society, from any exercise by You of the rights granted under this License that is for a purpose or use which is otherwise than noncommercial as permitted under Section 4(c).

f. Except as otherwise agreed in writing by the Licensor or as may be otherwise permitted by applicable law, if You Reproduce, Distribute or Publicly Perform the Work either by itself or as part of any Adaptations or Collections, You must not distort, mutilate, modify or take other derogatory action in relation to the Work which would be prejudicial to the Original Author's honor or reputation. Licensor agrees that in those jurisdictions (e.g. Japan), in which any exercise of the right granted in Section 3(b) of this License (the right to make Adaptations) would be deemed to be a distortion, mutilation, modification or other derogatory action prejudicial to the Original Author's honor and reputation, the Licensor will waive or not assert, as appropriate, this Section, to the fullest extent permitted by the applicable national law, to enable You to reasonably exercise Your right under Section 3(b) of this License (right to make Adaptations) but not otherwise.

5. Representations, Warranties and Disclaimer

UNLESS OTHERWISE MUTUALLY AGREED TO BY THE PARTIES IN WRITING AND TO THE FULLEST EXTENT PERMITTED BY APPLICABLE LAW, LICENSOR OFFERS THE WORK AS-IS AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE WORK, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR THE ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OF ABSENCE OF ERRORS, WHETHER OR NOT DISCOVERABLE. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO THIS EXCLUSION MAY NOT APPLY TO YOU.

6. Limitation on Liability. EXCEPT TO THE EXTENT REQUIRED BY APPLICABLE LAW, IN NO EVENT WILL LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES ARISING OUT OF THIS LICENSE OR THE USE OF THE WORK, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. Termination

a. This License and the rights granted hereunder will terminate automatically upon any breach by You of the terms of this License. Individuals or entities who have received Adaptations or Collections from You under this License, however, will not have their licenses terminated provided such individuals or entities remain in full compliance with those licenses. Sections 1, 2, 5, 6, 7, and 8 will survive any termination of this License.
b. Subject to the above terms and conditions, the license granted here is perpetual (for the duration of the applicable copyright in the Work). Notwithstanding the above, Licensor reserves the right to release the Work under different license terms or to stop distributing the Work at any time;

Bibliography

[1] Murrell, P. *R Graphics* (Chapman & Hall/CRC, Boca Raton, FL, 2005). ISBN 1-584-88486-X.
URL http://www.stat.auckland.ac.nz/~paul/RGraphics/rgraphics.html

[2] Sarkar, D. *Lattice Multivariate Data Visualization with R* (Springer, New York, 2007). ISBN 978-0-387-75968-5.

[3] Venables, W.N. and Ripley, B.D. *Modern Applied Statistics with S-PLUS*. Statistics and Computing (Springer-Verlag, 1999).

[4] Pinheiro, J.C. and Bates, D.M. *Mixed-Effects Models in S and S-PLUS*. Statistics and Computing (Springer-Verlag, 2000).

[5] Amit, O., Heiberger, R.M. and Lane, P.W. Graphical approaches to the analysis of safety data from clinical trials. *Pharm Stat* **7** (2008):20–35.