

Introduction to R and Basic Concepts from Probability and Statistics

Installation It is straightforward to install R on your Windows or Macintosh computer that is connected to the internet. Go to the web page <http://cran.r-project.org/> or its US mirror <http://cran.us.r-project.org/> and click on the link for your operating system and follow the instructions to download and install the latest version of R. (I recommend upgrading if your version is older than the most recent.) When I recently upgraded R for my Macintosh, installation required entering my password, but otherwise was following instructions and clicking with the mouse for default choices. Windows is similarly easy.

You only need to install R once. You may want to check back to the R web site every six months or so. New versions of R are released regularly.

Packages R is extended with many packages written by the large R community of developers. We will need several packages for this course. Installing packages with a computer connected to the internet is simple and is accomplished with a command typed into R when it is running. To install the `arm` package written for this textbook, follow these steps.

1. Start R.
2. At the command line, type:

```
> install.packages("arm")
```

Here is a complete list of packages we will need.

- `arm` has functions specific for this textbook;
- `lattice` has functions we will use for graphics;
- `lme4` has functions for mixed effects models;
- `Matrix` has functions that `lme4` requires. We will never use it directly.

Some of these packages may be installed automatically when you install others as there are dependencies. For example, `arm` requires `lattice` and `install.packages` may get `lattice` for you if you do not already have it when you install `arm`.

Good Practices It is wise to make a separate folder (directory) for this course with subdirectories for separate assignments or projects. For each assignment/project, it is good practice to keep a separate plain text document with a list of commented commands you can use to replicate what you have done to read in and set up a data set, carry out an analysis, and to create graphics. It is much wiser to keep a copy of the code to do something than the objects that you create.

It is also very wise to include comments in this file. Use the `#` character to create comments in the file. Any text after a `#` symbol is treated as a comment.

On the Macintosh, you can use the TextEditor application for this purpose. On Windows, Notepad is a good choice. My favorite editor is Emacs. There is also an editor you can use within R for this purpose. The Mac version has icons you can click on to open existing or new documents. I expect Windows does as well.

It is standard to give files of R commands (with comments, possibly) the extension `.R`. (Some operating systems may require three letter extensions. I do not know the default for these.)

You can use the `source` command to read in and execute an R command file.

Probability Distributions The three probability distributions used most often in biological applications are the binomial, Poisson, and normal. Here is a quick review of each.

The *binomial distribution* counts the number of “successes” in a fixed number n of independent trials (random events with two possible outcomes) that each has success probability p . The probability of exactly k successes is

$$\mathbb{P}(X = k) = \binom{n}{k} p^k (1 - p)^{n-k} = \frac{n!}{k!(n-k)!} p^k (1 - p)^{n-k}, \text{ for } k = 0, \dots, n$$

and the mean and standard deviation are $\mu = np$ and $\sigma = \sqrt{np(1-p)}$. Examples of biological variables suitable for the binomial distribution include the number of quadrats out of 25 that contain at least one plant of a given species, or the number of dogs with black coats in a litter of eight puppies.

The binomial distribution is the basis for statistical inferences about proportions as the sample proportion is $\hat{p} = X/n$. The mean and standard deviation of the sample proportion are p and $\sqrt{p(1-p)/n}$ respectively.

The *Poisson distribution* is another discrete probability distribution that arises in many biological applications. The Poisson distribution counts the number of events and can be derived from the three following assumptions: the expected number of events is proportional to the size of the region (in time or space), no two events can occur exactly at the same point, and regions with no overlap are independent. Unlike the binomial distribution, the Poisson distribution has no largest possible value. The probability distribution is defined by a single parameter μ which is both the mean and variance, so the standard deviation is $\sqrt{\mu}$. The probability of an outcome of k is

$$\mathbb{P}(X = k) = \frac{e^{-\mu} \mu^k}{k!}, \text{ for } k = 0, 1, \dots$$

The Poisson distribution can be used for the number of plants of a given species in a quadrat, the number of offspring in a given mating, or the number of seeds a plant produces, for example.

The *normal distribution* is the continuous distribution with the famous bell-shaped curve. It has a mean μ and standard deviation σ . Visually, the standard deviation is the horizontal distance from the point where the curve is steepest (either above or below the mean) to the mean. The normal distribution is the limiting distribution in the Central Limit Theorem — the distribution of sample means from large samples tend to be approximately normal even when the underlying population is not. More generally, the normal distribution arises from sums of large numbers of small random effects. Many biological quantitative traits are the result of a large number of small genetic and environmental effects which is why many (but certainly not all) such traits are well approximated by a normal distribution.

R has a four built in functions for each of these probability distributions (and others). The naming convention is to use a single letter prefix for the type of function and a stem for the distribution; here `binom` for binomial, `pois` for Poisson, and `norm` for normal. The prefix names make sense for continuous distributions but are used for discrete distributions also: “d” for “probability density” (probability mass for discrete distributions), “p” for the “cumulative probability” (I think `c` would have been a better choice), “q” for “quantile”, and “r” for “random”. The typical usage is that the first argument is the desired point and the subsequent arguments are parameters. For example, the probability of exactly 7 successes from 13 trials with a success probability of 0.6 is

```
> dbinom(7, 13, 0.6)
```

```
[1] 0.1967596
```

the probabilities of possible outcomes 0, 1, and 2 for a Poisson distribution with mean $\mu = 5$ are

```
> dpois(0:2, 5)
```

```
[1] 0.006737947 0.033689735 0.084224337
```

the area to the left of 93 under a normal curve with mean $\mu = 100$ and standard deviation $\sigma = 15$ is

```
> pnorm(93, 100, 15)
```

```
[1] 0.3203692
```

the cutoff points for the middle 80% of the same distribution are

```
> qnorm(c(0.1, 0.9), 100, 15)
```

```
[1] 80.77673 119.22327
```

and a random sample of 12 draws from a binomial distribution with parameters $n = 10$ and $p = 0.3$ are

```
> rbinom(12, 10, 0.3)
```

```
[1] 4 4 3 1 4 3 3 1 2 5 2 2
```

When using a random number generator in R for work that you want to replicate, it is a good idea to use the `set.seed` function and to save this call in your script. Random number generators in computers are actually *pseudo-random number* generators — they are a deterministic sequence of numbers that behave like random sequences. Setting the seed initializes the deterministic sequence. Resetting the seed to the same value ensures that a random simulation will turn out exactly the same. Here is an example with generating normal random variables.

```
> set.seed(3451)
```

```
> x1 = rnorm(5)
```

```
> x2 = rnorm(5)
```

```
> set.seed(3451)
```

```
> x3 = rnorm(5)
```

```
> x1
```

```
[1] 1.3235961 0.5081217 -0.2037756 0.2690817 1.2955921
```

```
> x2
```

```
[1] 1.7579618 -0.1068749 0.7928264 0.8622627 -0.3390764
```

```
> x3
```

```
[1] 1.3235961 0.5081217 -0.2037756 0.2690817 1.2955921
```

Reading in Data The object in R to store data is a *data frame* and the easiest way to create one is with the `read.table` command. The file you read in should be plain text (not a Word document) with “white space” (spaces, tabs) between values on each rows. Each row constitutes a single observation. The first row should be a *header* with the variable names. Categorical variables (factors) should use words, not numbers, for the levels. (For example, use “male” and “female”, not 0 and 1, to code for sex.) When you do use a header line, you need to specify it as in this example. Notice also that it is good practice to check the data you read in using the `str` command so that you can check the structure of the data frame.

```
> fevdata = read.table("fevdata.txt", header = T)
> str(fevdata)
```

```
'data.frame':      654 obs. of  5 variables:
 $ age  : int  9 8 7 9 9 8 6 6 8 9 ...
 $ fev  : num  1.71 1.72 1.72 1.56 1.90 ...
 $ ht   : num  57 67.5 54.5 53 57 61 58 56 58.5 60 ...
 $ sex  : Factor w/ 2 levels "female","male": 1 1 1 2 2 1 1 1 1 1 ...
 $ smoke: Factor w/ 2 levels "false","true": 1 1 1 1 1 1 1 1 1 1 ...
```

The variable names within the data frame are not known to R unless some action is taken. One choice would be to use the command `attach(fevdata)`. After typing this command, for the rest of the session, the names would be known. However, this is potentially dangerous as the names could be confused if they are also names of R commands or if they are in other data sets that you might subsequently attach.

Below I will show alternatives, either using the `with` command or including a `data` argument to a command for a graph or fitting a model.

Numerical Summaries R has many functions to compute numerical summaries of data. Here are examples to compute the mean and standard deviation of the age variable. Notice the use of `with` to specify the data frame where the `age` variable is located.

```
> with(fevdata, mean(age))
```

```
[1] 9.931193
```

```
> with(fevdata, sd(age))
```

```
[1] 2.953935
```

It can also be useful to compute summaries separately for subgroups. The `split` function splits a variable into groups according to the levels of a factor, creating a list with the observations from each level as the list elements and `sapply` will apply a function to each element in the list. Here are the average heights by sex.

```
> with(fevdata, sapply(split(ht, sex), mean))
```

```
female    male
60.21195 62.02530
```

Graphics R comes with a built set of graphics functions. Users interested in applying ideas from research in effective visual displays of data have created an entirely separate graphical system within R. This new system is generally incompatible with the old in that you cannot use commands from the old system to modify graphics made with the new, or vice versa. I will demonstrate how to use this new graphical system exclusively. Be aware that most of the examples you will see in other sources still use the old standard graphics system.

To begin, you need to have the `lattice` package. You only need to download and install this package once, and I will assume that you have done so. In each R session, however, you do need to load the `lattice` library.

```
> library(lattice)
```

The first function I will demonstrate is `histogram` which creates, not surprisingly, histograms. One difference between `lattice` and standard R graphics functions is that `lattice` functions create an object that you can save and update. Normally, you need to explicitly use a command to plot the object. This is in contrast to standard R graphics functions that produce a graph, but do not save anything. An exception is that `lattice` commands entered directly in the R console *will* produce the graph. So typing

```
> histogram( ~ age, data = fevdata)
```

directly will produce a graph but using `source` to read in a file with a line

```
histogram( ~ age, data = fevdata)
```

will have no effect. To make the plot when reading the file, you need to explicitly use a `plot` (or `print`) command.

```
plot( histogram( ~ age, data = fevdata) )
```

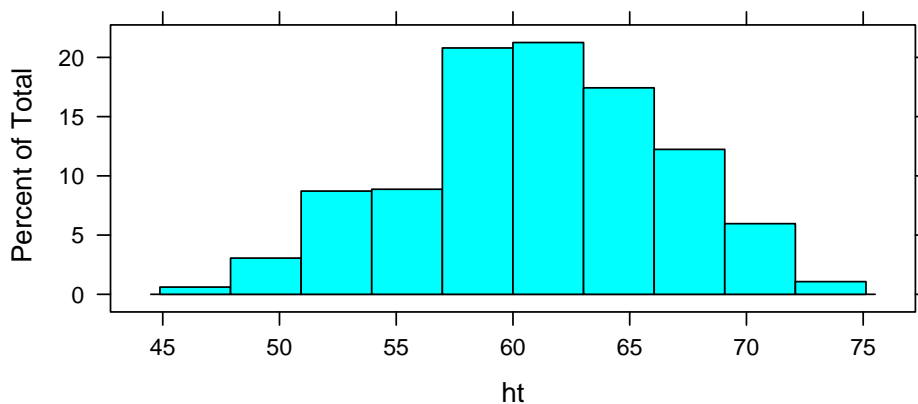
A peculiar feature of all `lattice` functions is that the first argument should be a *formula* that specifies the variables to be plotted. The syntax is

y-axis variable x-axis variable | conditioning variables separated by + or *

The y-axis variable is not used for plots such as histograms, but it is good practice to include the tilde (`~`) in any case. The bar is only used if there is one or more conditioning variables. I will show this usage later.

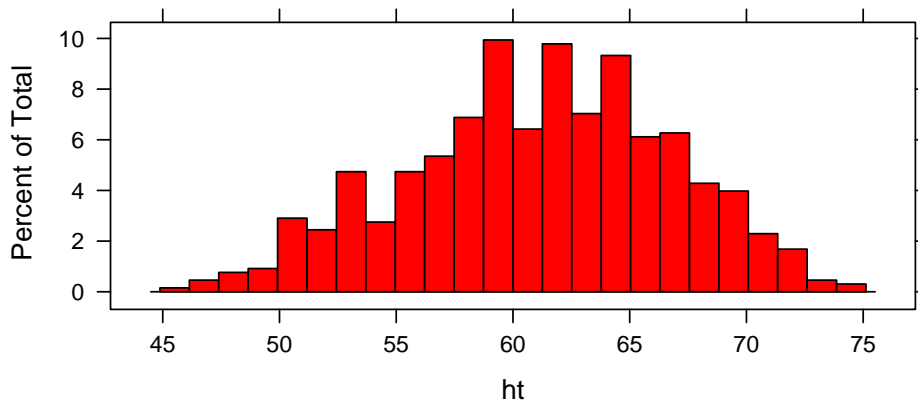
So, here is an example of a histogram of the heights of the children. The variable `ht` is in the data frame `fevdata`. I use the `data` argument to specify this. If I had used `attach` earlier, this would not be necessary.

```
> plot(histogram(~ht, data = fevdata))
```



If I decided that I wanted to change the color to red and increase the number of intervals to 24, I could do this.

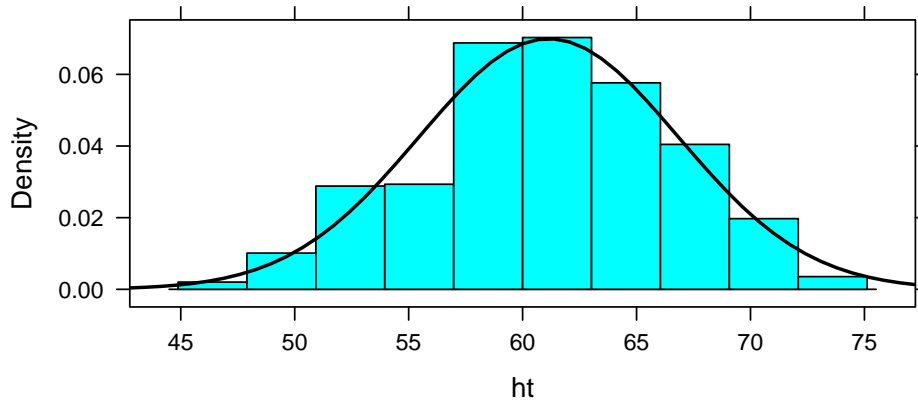
```
> plot(histogram(~ht, col = "red", nint = 24, data = fevdata))
```



Overlaying a curve such as a normal curve over a histogram is a more complicated procedure. The idea is to use the `panel` argument in `histogram` to define a new function to specify what is actually drawn. Do not worry about understanding this syntax — just copy the example when needed. My suggestion is to put the definition of the `overlayNormal` function in a file that you can read in when needed. You only need to define it once, and you can use it whenever you want after it is defined.

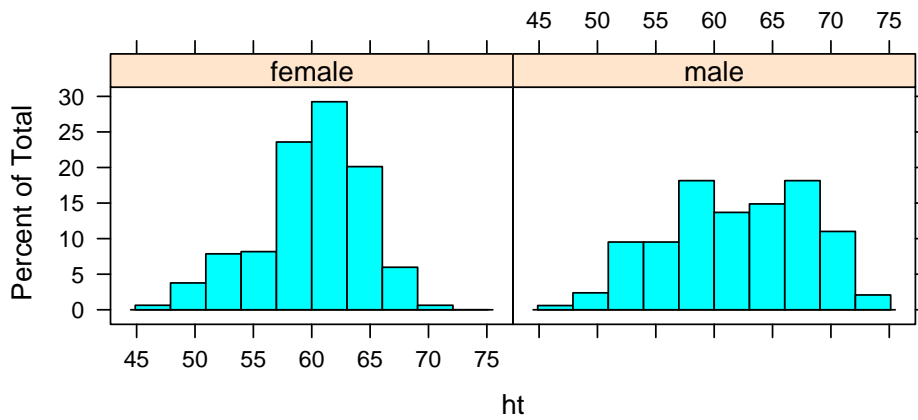
The argument `type="density"` tells `histogram` to scale the y-axis as a probability density (so the total area is one) instead of using counts.

```
> overlayNormal = function(x, ncol = "black", nlwd = 2, ...) {
+   panel.histogram(x, ...)
+   panel.mathdensity(dmath = dnorm, col = ncol, args = list(mean = mean(x),
+     sd = sd(x)), lwd = nlwd)
+ }
> plot(histogram(~ht, type = "density", panel = overlayNormal,
+   data = fevdata))
```



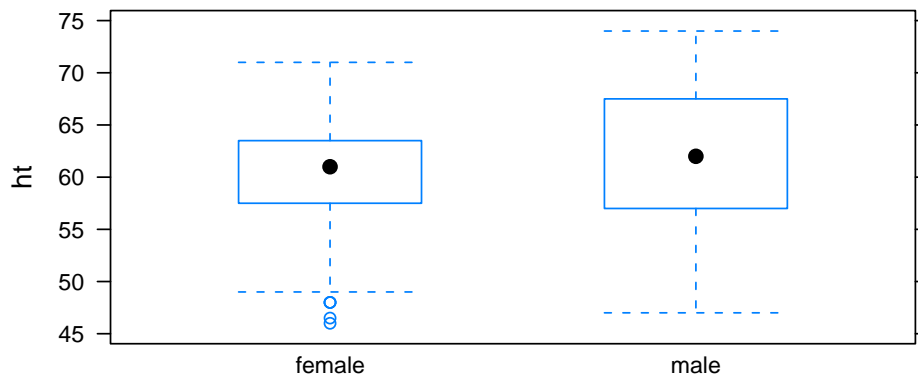
We can see separate histograms of height by sex using a conditioning variable.

```
> plot(histogram(~ht | sex, data = fevdata))
```



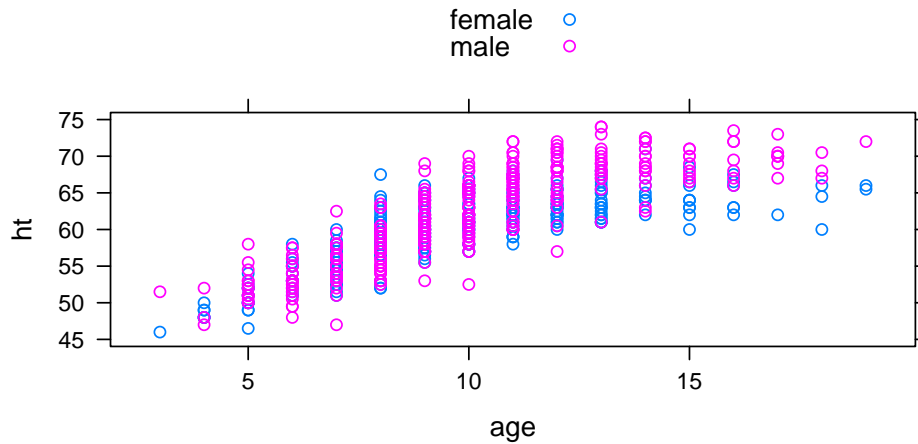
An alternative way to compare two distributions is with side-by-side box and whisker plots.

```
> plot(bwplot(ht ~ sex, data = fevdata))
```



The `xyplot` function makes scatter plots. Here `sex` is named as a group which puts all points in the same plot, but uses different symbols or colors. The `auto.key=T` argument automatically creates a key to the plotting colors or symbols used in the plot.

```
> plot(xyplot(ht ~ age, group = sex, auto.key = T, data = fevdata))
```



I will provide many more details on how to make nice graphs throughout the semester.