R notes for S&DS 220

2018-03-27

Contents

About

These notes organize and summarize the R material you need to know for S&DS 220.

Chapter 1

Basics

1.1 Setting up

R is a programming language that's popular for data manipulation. An up-to-date R interpreter can be downloaded from r-project.org. After downloading and installing the interpreter, you'll also want to download and install RStudio which provides a helpful visual interface for you to work with the interpreter.

1.2 Language fundamentals

Open RStudio, and find the box labeled **Console**. There should be a blinking `|` to the right of the `>` at the bottom of the box; if there isn't, then click to the right of `>`. Now type `1+2` and hit enter. The R interpreter reads your message and responds with `[1] 3`. Ignore the `[1]` for now; the R interpreter acted like a calculator and told us that 1+2 is 3. Next type `a <- 1+2` and hit enter. This time the result of the calculation `1+2` is stored as an object called `a`. Type `a` and hit enter to see its value.

Let's see one more quick example. Type `1:100` and hit enter. This is a convenient way to create a vector comprising the integers from 1 to 100.

Exercise: Try the command `rep(1, 100)`. What does it do? Looking at the interpreter's response to this command and to `1:100`, figure out what the `[1]` means.

Exercise: Make a guess at what R's response to the following commands will be. Then try them and explain what you think they're doing. (If you have no idea what to guess, just try it and see what happens.)

- `(1:5)^2`
- `sum(1:5)`
- `sum(rep(1, 5))`
- `mean(1:5)`
- `prod(1:5)`
- `max(1:5) - min(1:5)`
- `1:4 + 3:6`
- `2 * (1:4)`
- `1:4 * 3:6`
- `log(1:4)`
- `log10(1:4)`
- `sqrt(1:4)`

- `c(1, 3, 5)`
- `c(1:4, 1, 3, 5)`
- `v <- c(c(1, 3, 5), 1:4); length(v); v`
- `m <- matrix(1:20, nrow=5, ncol=4); m`
- `rowSums(m)`
- `ignore this`
- `# ignore this`
- `print("hello")`
- `paste("the square root of 25 is", sqrt(25)); print("hello")`
- `cat("the square root of 25 is", sqrt(25)); print("hello")`
- `cat("the square root of 25 is", sqrt(25), "\n"); print("hello")`

An important feature of R is the ability to easily select from a vector or matrix. Try the following (make sure you've run the code from the exercise above):

- `v; v[3]; v[3:5]; v[c(1, 2, 1)]; v[-3]`
- `v[2] <- 100; v`
- `m; m[1, ]`
- `m[, 2:3]`
- `m[1:2, c(2, 4)]`

1.2.1 Functions

Both `rep` and `sum` are functions that are built-in to R, but we can also write our own functions. Give the console the following command.

```
sumsq <- function(v) { return(sum(v^2)) }
```

This creates a *function* object called "sumsq" which is now available for us to use on any numerical vector (which the function will call "v" internally). It squares the vector's entries, adds up those squared values, then gives us back the result of that calculation.

Exercise: Predict the response to `sumsq(1:3)`, then try it out to check your prediction.

In practice, we won't usually type our code directly into the console. Rather, we will type it into a text file so that we can more easily keep track of what we've done. From the RStudio menu, select File > New File > R Script to create an empty file. Enter the following code:

```
sumsq <- function(v) {
  return(sum(v^2))
}

avgsq <- function(v) {
  return(sumsq(v)/length(v))
}
```

You've already defined `sumsq` and R remembers what it means, but now you've also recorded that definition so that you can easily check it, change it, or share it with others. We still need to tell the interpreter what we want `avgsq` to mean. Click and drag your cursor across the three lines of code that define `avgsq`. Then in the RStudio menu, find Code > Run Selected Line(s). There should be symbols next to "Run Selected Line(s)" that tell you what its hotkey is (e.g. on a Mac, it's Command+Enter.) Take note of this hotkey, because you'll use it constantly when working in RStudio. Press the Run hotkey once to run the three lines of code that define `avgsq`.

Exercise: The `avgsq` function calculates the average of the squared values of a vector. Test this by running `avgsq` in the console. Suppose that what we really wanted it to do was calculate the square of the average value of a vector. Change the code so that it does this, then run `avgsq(1:2)` in the console. It gives the same number as before because the console isn't aware that you've changed the definition of `avgsq`. You have to tell it. Select the code defining `avgsq` and press the hotkey to run it. Then run `avgsq(1:2)` in the console one more time to check that your new code is working as expected.

1.2.2 Conditionals

Some expressions are interpreted as being either `TRUE` or `FALSE`. Run the following commands one at a time, and think about their output.

```
a
a <= 12
a > 4
a == 4
a == 3
a != 3
a != 4
```

The `!` symbol means *not*. Notice that checking for equality involves *two equals signs*.

An *if* statement only runs its code if the expression between the parentheses is `TRUE`. Type the following in the bottom of your file, then run it.

```
if(a < 3) {
  print("It's strictly less than 3")
}

if(a <= 3) {
  print("It's less than or equal to 3")
}
```

A `TRUE`/`FALSE` vector can also be used to select a subset from another vector. Try the following:

```
v <- 1:20
v > 12
v[v > 12]
```

1.2.3 Loops

Type the following at the bottom of your file, then run the code.

```
for(i in 1:3) {
  print('hello')
}
```

This is a *for* loop. The part between `{` and `}` is called the *body* of the loop. For each element in the vector `1:5`, the interpreter runs the body of the loop. The the loop is interpreted to mean

```
i <- 1
print('hello')

i <- 2
print('hello')
```

```
i <- 3
print('hello')
```

Because `i` doesn't show up in the body, the loop does the exact same thing each time through. Change the code as follows, and run it again.

```
for(i in 1:3) {
  print(rep("hello", i))
}
```

When the number of iterations needed isn't known ahead of time, a *while* loop is a good alternative to a *for* loop. The body of the *while* loop runs over and over as long as the expression inside the parentheses is `TRUE`. Type and run the following:

```
i <- 1
while(2^i <= 1000) {
  print(2^i)
  i <- i + 1
}
```

Another way to end a loop is by using `break`. The following code block works just like the code above.

```
i <- 1
while(T) {
  print(2^i)
  i <- i + 1
  if (2^i > 1000) {
    break
  }
}
```

Note that the R interpreter accepts `T` and `F` to mean `TRUE` and `FALSE`.

1.2.4 Workspace

Press the Save hotkey (look at File > Save to see what the hotkey is). Enter the name "example" and save the file to your Desktop folder. You have created the file "example.R" in your Desktop folder.

From the RStudio menu, select Session > Clear Workspace. This removes the objects you've defined so far. Try `sumsq(1:3)` in the console, and you will get an error.

Next click Session > Set Working Directory > Choose Directory. Select your Desktop folder (it may already be selected), and press Enter. Run the command `source("example.R")` in the console; this command tells the interpreter to read in the entire contents of the R Script file *example.R* which it expects to find in the current *working directory*. Run `sumsq(1:3)` in the console to verify.

Finally, click the tiny "x" to the right of where it says "example.R" in RStudio to close the file. In a file browser, delete the file *example.R*; we won't be needing it anymore.

1.3 R packages

So far, you've been using "base R." It has a variety of built-in functions like `sum` that make it a relatively convenient language for data analysis. Our `sumsq` and `avgsq` functions are built "on top" of base R and they are tools that might make some data analysis tasks ever-so-slightly more convenient. In fact, over the past decades, countless R users have been writing code that they find convenient and sharing it with the world by putting it in an R *package*. A few packages are automatically installed along with the R interpreter

when you download it; one of these is the *MASS* package. Try the following commands in the console in this order:

- `Traffic`
- `MASS::Traffic`
- `Traffic`
- `library(MASS)`
- `Traffic`

There is an object called "Traffic" stored in the *MASS* package. It can be accessed by either reaching selectively into *MASS* (with `MASS::Traffic`) or after dumping the entire contents of *MASS* into your *environment* a.k.a. *workspace* (with `library(MASS)`).

The vast majority of the publicly available R packages need to be downloaded before they can be used. Over ten thousand of those packages (including the most commonly used ones) are hosted by a group called the Comprehensive R Archive Network (CRAN). Those packages can be easily downloaded right from the console. As an example, use `install.packages("rmarkdown")` to download the *rmarkdown* package; any other packages that *rmarkdown* uses will also be installed automatically by this command.

Exercise: To be more precise, you just installed *the current version* of the *rmarkdown* package. Look at the interpreter's output and figure out what version number of *rmarkdown* was installed.

1.4 More

We've covered some of the very basics here; you will learn much more as we go. A nice *cheat sheet* summarizing R fundamentals is available here.

If you want more information about any object (e.g. a function or a dataset) in base R or in a package that you've loaded into your environment, click on RStudio's **Help** tab, type the name of the object, and press enter. Related commands are often bundled together into the same help file.

Exercise: Read about the "file" argument in the help page for `read.csv`; the file's location can be specified with either a path on your computer or a url. Run the following code to read in a csv file from the web.

```
d <- read.csv("http://www.stat.yale.edu/~jtc5/220/data/CEO_Salary_2012.csv")
```

Then download the file "CEO_Salary_2012.csv" onto your computer and use `read.csv` again to read it in locally.

You can also search the web for answers to your questions and if necessary post to StackOverflow.

Chapter 2

Plotting

2.1 Histograms

The histogram is a nice tool for roughly visualizing the distribution a numerical vector.

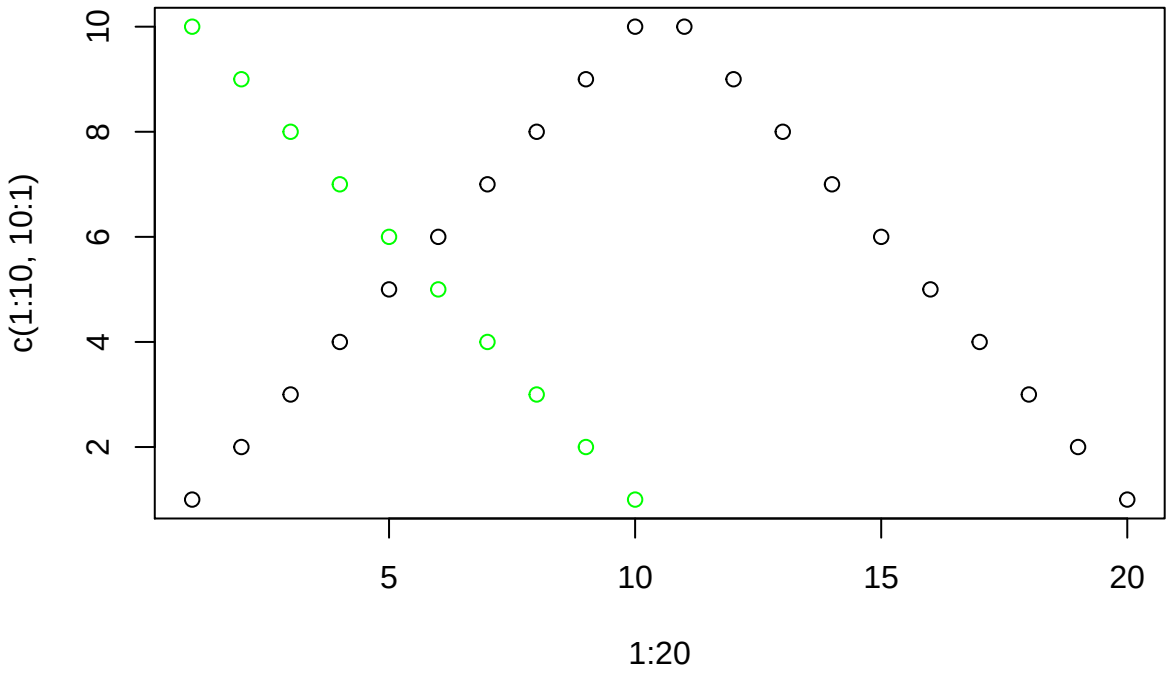
Exercise: Run the following code, then look at the help files for `hist` and `truehist` (which is in the MASS library). Describe their differences.

```
library(MASS)
class(Traffic)
names(Traffic)
Traffic$y
hist(Traffic$y)
truehist(Traffic$y)
```

2.2 Scatterplots

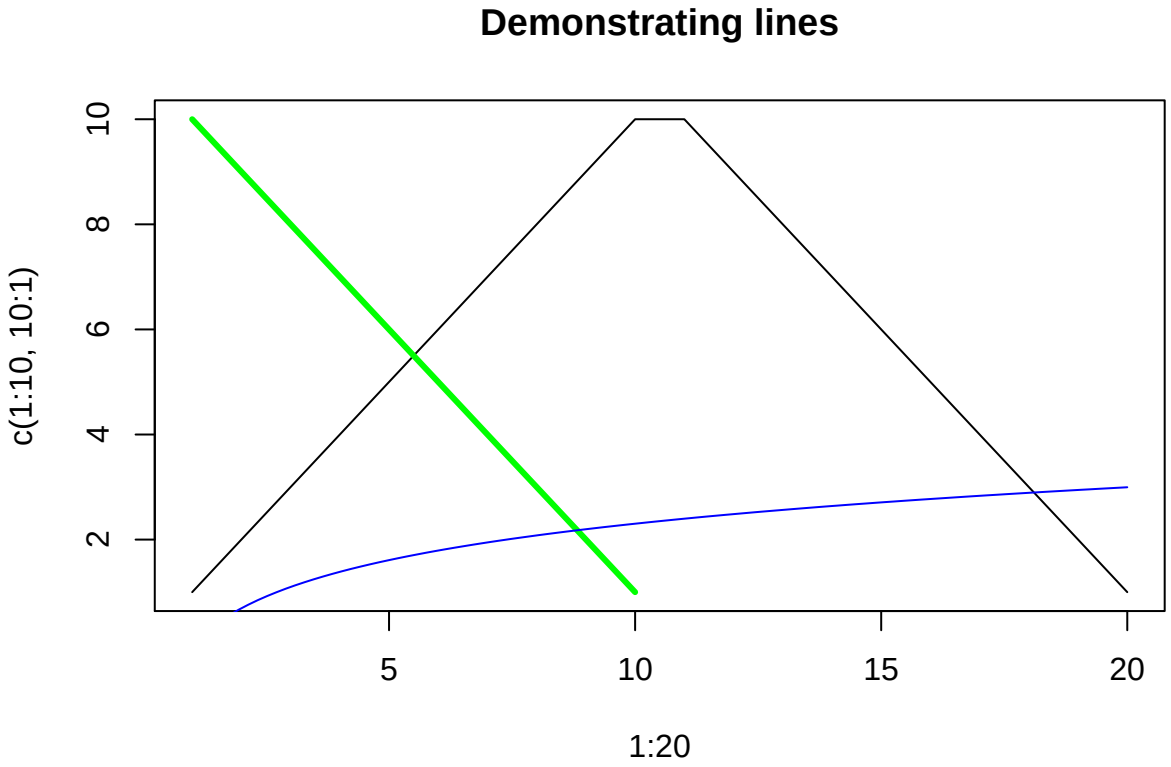
Two vectors can be put together into a scatterplot using the `plot` command. The `points` command adds points to an existing plot. The `col` attribute allows you to set the color of the points (used in either `plot` or `points`).

```
plot(1:20, c(1:10, 10:1))
points(1:10, 10:1, col="green")
```



To draw connected line segments instead of points, use `type="l"`; to add line segments to an existing plot, use `lines` (for a pair of vectors) or `curve` (for a function).

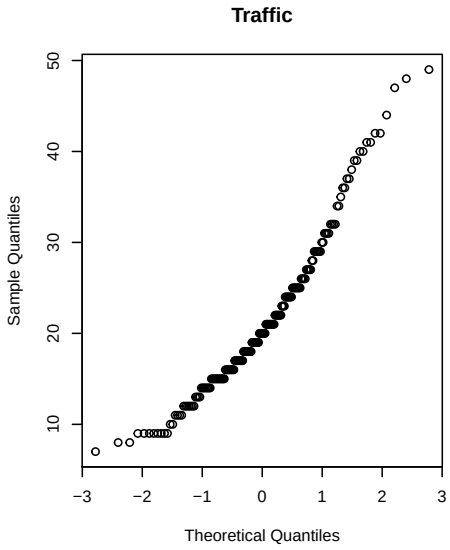
```
plot(1:20, c(1:10, 10:1), type="l",
     main="Demonstrating lines")
lines(1:10, 10:1, col="green", lwd=3) # lwd sets line width
curve(log(x), col="blue", add=TRUE)
```

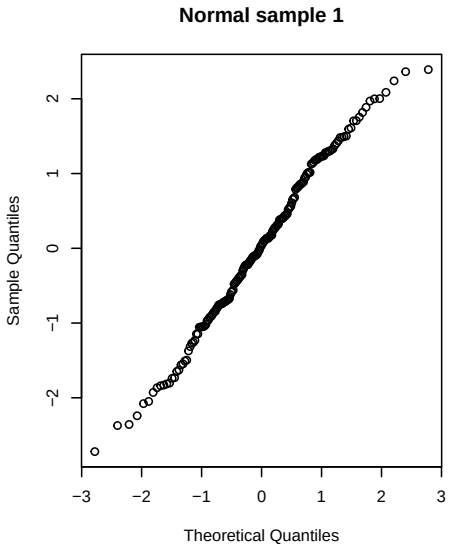


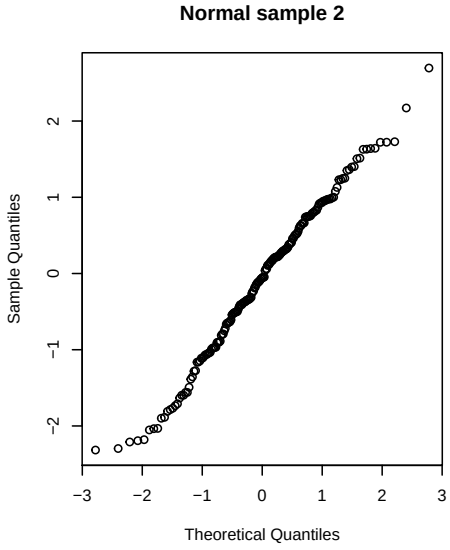
The distribution of a numerical vector can be checked for normality using a *Normal quantile plot* which is a scatterplot of the vector's ordered values paired with idealized ordered values of a truly Normal sample.

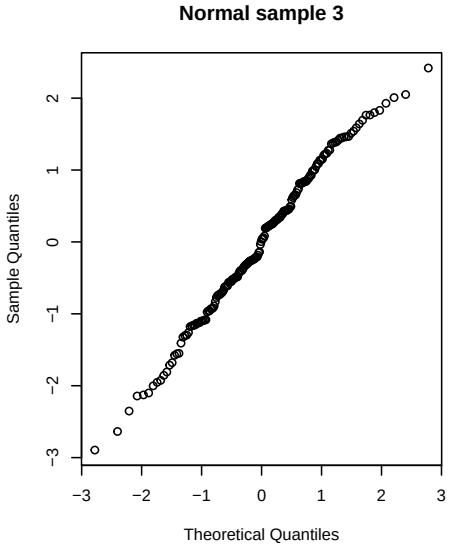
The R command is `qqnorm`; for example:

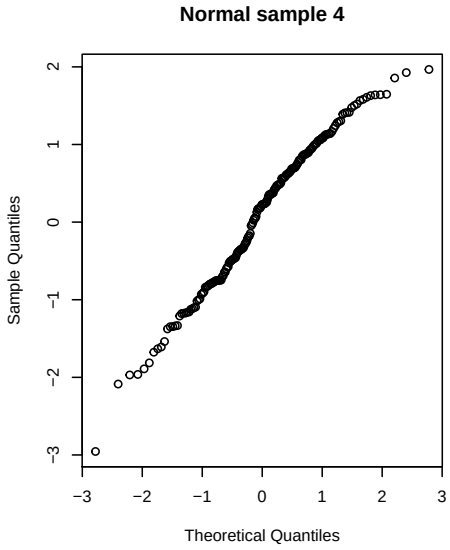
```
y <- MASS::Traffic$y
par(mfrow=c(3, 2))
qqnorm(y, main="Traffic")
for(i in 1:5) {
  qqnorm(rnorm(length(y)), main=paste("Normal sample", i))
}
```

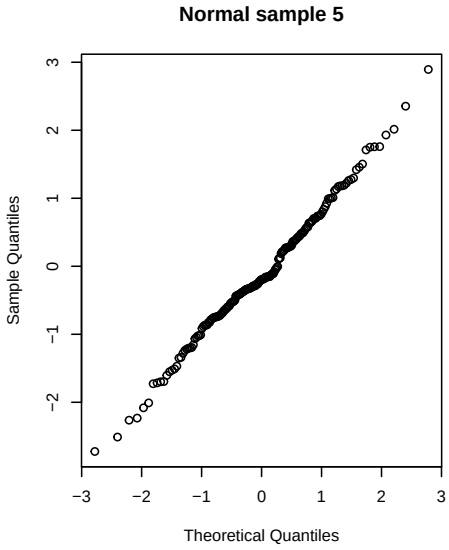

Traffic
Sample Quantiles
Theoretical Quantiles


Normal sample 1
Sample Quantiles
Theoretical Quantiles


Normal sample 2
Sample Quantiles
Theoretical Quantiles


Normal sample 3
Sample Quantiles
Theoretical Quantiles


Normal sample 4
Sample Quantiles
Theoretical Quantiles


Normal sample 5
Sample Quantiles
Theoretical Quantiles

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

The `par(mfrow=c(3, 2))` command told the interpreter to place the next six plots together in a 3 by 2 grid.

Chapter 3

Markdown

A language called *Markdown* is commonly used to indicate simple structure and formatting within a plain text file without cluttering it. The text is processed by a parsing program which turns it into a structured and formatted document. As a very simple example, "this *Markdown* text" gets displayed as "this *Markdown* text"; text that lies between asterisks gets italicized. Markdown also makes it easy to make bold text, create links, create tables, insert images, write LaTeX math, and the list goes on. See RStudio's guide for more.

We will use a language called R Markdown to write nice polished documents that fluidly incorporate R code and plots. From the RStudio menu, select File > New File > R Markdown. Make the title "Nice Example" and click OK. RStudio opens a simple example R Markdown (Rmd) file. Go to File > Knit Document (and make note of the hotkey); RStudio will prompt you to save the file, then it will display the processed example document as a webpage. In the file editor, change the first sentence under "## R Markdown" to "This is my *first* R Markdown document." Then press the Knit Document hotkey again to see your change implemented.

R Markdown does ordinary markdown parsing, but it does a bit of extra processing first. It looks for R code to run; notice what happens with `summary(cars)` and `plot(pressure)` from the Rmd file. R Markdown displays your R code neatly, along with the interpreter's output and plots.

Exercise: Change "{r pressure, echo=FALSE}" to "{r}" and knit the document again. What do you think "pressure" and "echo=FALSE" were doing?

Exercise: At the top of the Rmd file, change "html_document" to "pdf_document" then Knit Document again.

The package knitr is used for specifying how you want to handle each code chunk, for instance, if you want to change the size of a plot.

Glance over this beginner's guide to get an idea of what else R Markdown can do.

Chapter 4

Probability

4.1 Sampling from a vector

The `sample` command draws a specified number of entries from a vector. If you want to randomly pick three (numbered) students from a class of ten, you can run `sample(1:10, 3)`.

Sometimes you want to draw *with replacement*, meaning that each pick ignores what has happened previously; after each draw, you *replace* your pick before drawing again. For instance, if you want to randomly assign twenty tasks to four employees, you could run `sample(1:4, 20, replace=TRUE)`. If you want employee number 1 to have twice as large of a probability as the other employees of being assigned each task, you'll need to use the `prob` parameter: `sample(1:4, 20, replace=TRUE, prob=c(2, 1, 1, 1))`.

4.2 Four basic functions for common distributions

For several families of distributions, the R language has four built-in functions corresponding to:

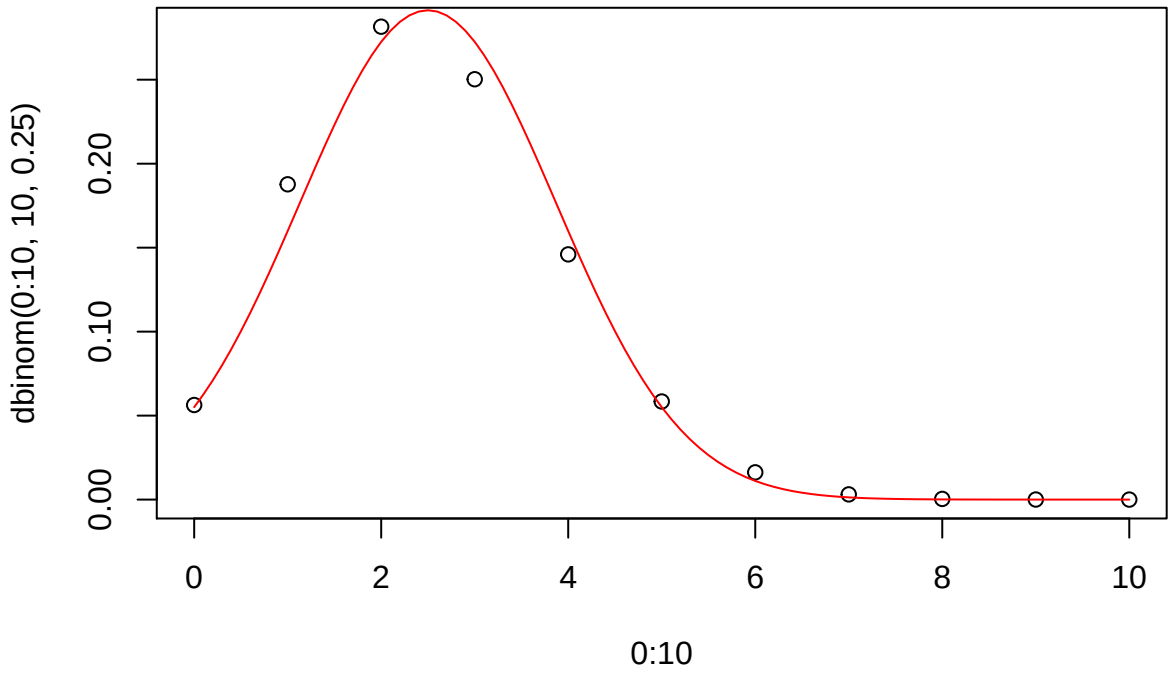
- Density (pdf for continuous distributions, pmf for discrete distributions)
- Probability less than or equal (cdf)
- Quantile (inverse cdf)
- Random sample

The function names follow the pattern "d," "p," "q," or "r" followed by an abbreviation for the family of distributions. For example, the probability of getting four heads out of ten flips of a coin with heads-probability .25 is calculated by the R command `dbinom(4, 10, .25)`. The probability of four or fewer heads is `pbinom(4, 10, .25)`. Next, `qbinom(.95, 10, .25)` tells you the smallest number of heads for which there is no more than .05 probability of exceeding. Finally, `rbinom(100, 10, .25)` simulates one hundred experiments of the twenty coin flips and tells you how many heads it got in each trial.

For Normal distributions, the abbreviation is "norm." If no additional parameters are specified, it assumes you want mean zero and standard deviation one. Otherwise use the second and third parameters (named "mean" and "sd").

Binomial distributions are discrete whereas Normal distributions are continuous. The four functions work in both cases.

```
plot(0:10, dbinom(0:10, 10, .25))
curve(dnorm(x, mean=10*.25, sd=sqrt(10*.25*.75)), from=0, to=10, add=TRUE, col=2)
```



Another family built in to R is the uniform distributions, abbreviated "unif." In particular, `runif(n)` generates `n` independent draws from the Uniform(0, 1) distribution.

Before asking the R interpreter to perform a random draw, you might want to use the `set.seed` command with any integer as the input. This sets a "starting point" for the random draw so that the exact same "random" results can be replicated by the code at a later time or by another computer.

```
runif(5)
```

```
## [1] 0.67451841 0.28798486 0.62727115 0.03874901 0.54447534
```

```
set.seed(1)
runif(5)
```

```
## [1] 0.2655087 0.3721239 0.5728534 0.9082078 0.2016819
```

When you run the above code, my first `runif(5)` output will differ from yours, but our draws will agree after we've both set the seed to 1.

4.3 Simulation

In a room of 100 people, is it likely that you will be able to find a birthday shared by two people? What about three people? Four? Let's pretend that leap-years don't exist, that each day of the year is equally likely to be a person's birthday, and that the people's birthdays are independent. Now we draw random birthdays accordingly.

```
n <- 100
set.seed(4)
b <- sample(1:365, n, replace=TRUE)
b
```

```
##   [1] 214   4 108 102 297  96 265 331 347  27 276 105  37 349 152 167 355
##  [18] 214 352 279 261 364 185 179 237 304 176 308 188 194 207  88 321 239
##  [35] 177 355 168 228 142   3 343  89 207  67 331  31 329 326 265 207 142
##  [52] 273 327 296 299 154  65  64 326 272 205  27 312 334  83 230  26 188
```

```
##  [69] 294 355 126 231 150 127 302 252 118 162  96  49 333 259 209 335 330
##  [86]  22  17 361  76 340  71  47 197 100 130  31 280 163  14 256
```

```
table(b)
```

```
## b
##   3   4  14  17  22  26  27  31  37  47  49  64  65  67  71  76  83  88
##   1   1   1   1   1   1   2   2   1   1   1   1   1   1   1   1   1   1
##  89  96 100 102 105 108 118 126 127 130 142 150 152 154 162 163 167 168
##   1   2   1   1   1   1   1   1   1   1   2   1   1   1   1   1   1   1
## 176 177 179 185 188 194 197 205 207 209 214 228 230 231 237 239 252 256
##   1   1   1   1   2   1   1   1   3   1   2   1   1   1   1   1   1   1
## 259 261 265 272 273 276 279 280 294 296 297 299 302 304 308 312 321 326
##   1   1   2   1   1   1   1   1   1   1   1   1   1   1   1   1   1   2
## 327 329 330 331 333 334 335 340 343 347 349 352 355 361 364
##   1   1   1   2   1   1   1   1   1   1   1   1   3   1   1
```

```
max(table(b))
```

```
## [1] 3
```

The `table` command scans through a vector and counts the number of occurrences of each value. The `max` tells us that there is at least one birthday shared by three people but no birthdays are shared by four or more.

But this is a single run of the experiment. To get a better idea of the range of possible outcomes and how probable they are, we want to run the experiment a large number of times.

```
nsim <- 10000
results <- rep(NA, nsim)
for(i in 1:nsim) {
  b <- sample(1:365, n, replace=TRUE)
  results[i] <- max(table(b))
}
table(results)/nsim
```

```
## results
##      2      3      4      5      6
## 0.3598 0.5786 0.0586 0.0029 0.0001
```

Exercise: The following simulation code runs fine, but it doesn't perform a valid simulation. Identify the TWO flaws and fix them.

```
n <- 100
nsim <- 10000
results <- rep(NA, nsim)
for(i in 1:nsim) {
  set.seed(3)
  results <- max(rnorm(n))
}
```

Run your fixed version of the simulation and plot a `truehist` of the results.

Chapter 5

Statistics

5.1 Binomial statistics

Suppose a casino game involves flipping a coin that they claim is fair, i.e. that heads and tails are equally probable. The casino wins the bet when which tails is flipped, while you win if it's heads. You play the game 1000 times and observe only 429 heads. Should you doubt the casino's claim that the coin is fair? We could use `pbinom` to calculate the exact significance probability, but let's instead try `prop.test` which uses the Normal approximation with a continuity correction to give a significance probability and confidence interval. (Add the parameter `correct=F` to suppress the continuity correction.)

```
prop.test(429, 1000, alternative="less")
```

```
##
##  1-sample proportions test with continuity correction
##
## data:  429 out of 1000, null probability 0.5
## X-squared = 19.881, df = 1, p-value = 4.121e-06
## alternative hypothesis: true p is less than 0.5
## 95 percent confidence interval:
##  0.0000000 0.4554038
## sample estimates:
##     p
## 0.429
```

If the coin really is fair, there's less than a one in ten thousand chance of an outcome at least this small. If the casino had instead claimed that the coin had heads probability of .48, you could test that by adding the parameter `p=.48` to the function call.

To test the other extreme with `prop.test`, use `alternative="greater"`; a two-sided test is used by default if you do not specify a value for `alternative`.

There are a variety of opinions about how to calculate confidence intervals for a Binomial success probability. A number of methods are provided by `binom.confint` in the `binom` package. (Do `install.packages("binom")` before running the code below.)

```
library(binom)
binom.confint(429, 1000, conf.level=0.95, methods="all")
```

```
##           method   x    n      mean     lower     upper
## 1  agresti-coull 429 1000 0.4290000 0.3986523 0.4598911
## 2     asymptotic 429 1000 0.4290000 0.3983243 0.4596757
## 3          bayes 429 1000 0.4290709 0.3984722 0.4597491
```

```
## 4        cloglog 429 1000 0.4290000 0.3981561 0.4594475
## 5          exact 429 1000 0.4290000 0.3980720 0.4603472
## 6          logit 429 1000 0.4290000 0.3986339 0.4599102
## 7         probit 429 1000 0.4290000 0.3985686 0.4598596
## 8        profile 429 1000 0.4290000 0.3985352 0.4598282
## 9            lrt 429 1000 0.4290000 0.3985450 0.4598228
## 10     prop.test 429 1000 0.4290000 0.3981599 0.4603924
## 11        wilson 429 1000 0.4290000 0.3986535 0.4598899
```

Exercise: Run a two-sided `prop.test` to see which of the `binom.confint` methods it agrees with.

5.2 More to come

Bibliography