# Statistical Programming 2024

## Contents

# 1 Software Requirements: R, git, JAGS etc

To complete this course you will need to get yourself a free github account and install the following software on your computer. Only students on *Extended* Statistical Programming need the additional software indicated by *Extended:*.

- R or Rstudio, git. *Extended:* pandoc and JAGS.
- R packages: debug, ggplot2. *Extended:* rjags, rmarkdown.
- *Extended:* Only if you do not already have latex installed, install tinytex.

It is assumed that you have the basic computer skills to do this. If not, you will need to spend some time online acquiring them (this course is about programming, not basic computer skills). If you have difficulty, you can post questions on piazza for other students to answer. Please answer other student's questions on installation on piazza.

In more detail.

- `R` is the free statistical programming language and environment that we will use. You can get a copy from the Comprehensive R Archive Network (CRAN)
  `https://cran.r-project.org/`
  just follow the links under 'Download and Install R'.
- `Rstudio` is an alternative front end for R, which some people prefer to use. It provides an R code editor, R session window, R graphics window and other information, all in one 'integrated environment'. If you prefer this to R as available from CRAN, you can get it from
  `https://posit.co/download/rstudio-desktop/`
- `git` is a version control system. It helps you to write code in teams, as you must do for this course, without breaking each other's work. It also lets you keep a record of the changes you make, so that you can go back to earlier versions, if you need to. To install `git` follow the instructions at
  `https://www.git-scm.com/`
  You do not need a GUI client. On Windows use the default options. `linux` systems often have it installed already and something like `sudo apt install git` will install it if not.
  See `https://training.github.com/` for further information.

*Extended:*

- `pandoc` is document conversion software used by the `rmarkdown` package. Installation instructions:
  `https://pandoc.org/installing.html`
- `JAGS` stands for 'Just Another Gibbs Sampler'. We will use it for programming Bayesian models and sampling from them. Downloads are available here:
  `https://sourceforge.net/projects/mcmc-jags/files/JAGS/4.x/`
  but under linux something like `sudo apt install jags` can be used.

Once the above are installed, you will need to install some packages from the R session command line. R/Rstudio can install packages in a local directory/folder for you, but I tend to start R as administrator/super user/root and then install them (e.g. on linux I would start R from the command line as `sudo R`). Here are the packages you need:

- Firstly we need a debugging package, to help you to find errors in your code, or understand how code is working by stepping through it. What is built into R and Rstudio is not great, and a much better option is Mark Bravington's debug package. This is not available from CRAN, but only from Mark's repository, so to install, you need to do this:

  ```
  options(repos=c("https://markbravington.github.io/Rmvb-repo",
        getOption( "repos")))
  install.packages("mvbutils")
  install.packages("debug")
  ```

  `debug` requires the `tcltk` package, which is installed by default in R, but has requirements which are not met by default on all mac installations. If installation fails for this reason you need to read the error message carefully - it will probably tell you to install `Xquartz`. On Windows the package may install, but then produce a code window with nothing in it if you try use it. Minimizing the window and then maximizing it fixes the problem - see `?debug-package` under "Display Woes" for a less annoying fix.

- `ggplot2` provides a nice alternative to built in R graphics. Install by typing
  `install.packages("ggplot2")`
  at the R command line.

*Extended:*

- `rjags` lets you use JAGS directly from R. Install using
  `install.packages("rjags")`
  once installed you should try it by typing `library(rjags)` within R, and preferably running the first example from the helpfile `?jags.samples`.

- `rmarkdown` provides a neat way of documenting data analyses using R. Install using
  `install.packages("rmarkdown")`

- `rmarkdown` requires a latex installation in order to allow you to produce pdf documents. So, *only if you do not already have latex installed*, install `tinytex` with `install.packages("tinytex")` then if you are *really sure* that you do not already have latex installed run `tinytex::install_tinytex()` WARNING: this command may break existing latex installations!

Once you have the software installed, you need to do one more thing. Get a free `github` account (if you do not already have one). `https://github.com/join` is where to get one.

## 1.1 Using a terminal window, choose a text editor

This course involves programming - that is writing text instructions that get a computer to do something. Anyone who programmes rapidly discovers that writing text to get a computer to do something is often quicker, more convenient and more reproducible than clicking your way through a graphical interface. This is sometimes true even for basic tasks such as file copying, or moving between directories/folders.

Whatever operating system you are using you will be able to launch a 'terminal window' for this purpose. Make sure that for your operating system you know how to do this, how to list the names of the files in a directory/folder, how to change from one directory/folder to another, how to delete a file, and how to close the terminal window.

You will also need to edit text files containing the computer code you write. If you choose to use `Rstudio`, there is a built in editor, and you might choose to just use that. If you use plain `R` then you will need an external editor. `Word` is not suitable, as it will insert hidden characters in your code that will cause problems, so you instead should use something simpler (e.g. `wordpad` or `notepad` on windows). Make sure you know what is available on your computer and use that. I tend to use `emacs`, but this has more features than you actually need for this course.

# 2 git and github

`git` is a system that lets you track changes in code files and to manage those files when several people are working on them. In particular it lets you maintain a central *repository* of your code, which serves as the backed up master copy. The repository is just a folder/directory containing the master copy of your code. Several people can make working local copies of the repository (on their own computers) and work on the code, only merging those changes into the master copy when they are ready. `git` provides the tools to allow management of the master copy in a way that avoids or resolves conflicts. A conflict is when two people made contradictory changes to the code.

## 2.1 Setting up a repo on `github`

An easy way to set up a central repository is to use `github`.

- Login to your `github` account on `github.com`, click on `Repositories` and then click on `New` to set up a new repository.
- Follow the instructions. You probably want your repository to be private if it is for working on assessed coursework.

For group homework, one person sets up the repo and invites their team mates to collaborate on it[1].

- Log on to `github`, nagivate to your repo, and click on `Settings`.
- Click on `Collaborators` from the left hand menu - you will need their github user name.
- Note that collaborators have to accept the invitation before they can save work (push) to the github repo – this applies whether the repo is private or public.

You can add files directly to your repo, via the web interface, but we will shortly cover how to add files to the local copy of your repo, and then *push* them to github.

## 2.2 Using `git`

Mostly, you will interact with your `github` repo via the `git` software, and your local copy of your `github` repo. You will use `git` by typing commands in a terminal window. To check everything is working before using `git` for the first time you might open a terminal window and check the `git` version, as follows:

```
git --version
```

This gives the answer `git version 2.25.1` for me. Here we will cover only the most basic use of `git` and `github`: the minimum needed to collaborate on projects and keep your work backed up. You can find much more at:

- `https://git-scm.com/docs/gittutorial`
- `https://guides.github.com/`
- `https://training.github.com/`

Note that `Rstudio` offers built in facilities for using `github`, which you may find useful. However before using them it is better to first learn how `git` and `github` work by using them in the way described here, which is transferable to any project, not just R programming with `Rstudio`.

Before moving on, if you are using a Mac or linux you probably want to issue the command:

```
git config --global core.autocrlf input
```

which deals with the fact that line ends are dealt with differently by different operating systems and handles this gracefully (Windows is set to do this by default).

[1] To stop participating in a repo, click on your picture at top right of github home page, select settings and then Repositories - you get a list of repos and can leave any of them.

### 2.2.1 Making a local working copy of the repo

After creating your github repo you need to make a local copy of it on your computer.

- From your `github` repo page (at `github.com`) click on `Code` select `Clone` and copy the `htpps` address for the repository that appears. For example, an outdated repo containing these notes has the address `https://github.com/simonnwood/sp-notes.git`
- In the terminal window on your local machine change directory (`cd`) to the folder where you would like the local repo to be located. Then type `git clone` followed by the address you copied. e.g.

  ```
  git clone https://github.com/simonnwood/sp-notes.git
  ```

A local copy of the repo is then created on your computer.

## 2.3 Modifying work and synchronising with the github repo

The local copy of your repo is just a directory/folder containing the files in your repo, and a hidden `.git` subdirectory that `git` uses to maintain an index of the files that it should keep track of, and the history of changes made. `git` knows about all the files in the `github` repo you cloned, but will only start keeping track of other files that you may add to your local repo when you tell it to do so (with `git add`).

Similarly `git` does not keep a record of all the changes you make to your code as you work on it. Rather, it takes and stores a 'snapshot' of the state of the files it is tracking only when you tell it to, using a `git commit` command.

Neither does `git` automatically save all these snapshots to your master `github` repo. That only happens when you tell it to using `git push`. This is quite useful, you can keep a detailed record of the changes you make while working on code locally, without modifying the `github` master repo until you are satisfied that the changes are complete and working.

Note that you can access the help pages for the main `git` commands by typing `git help` followed by the command name. For example, if you want to know more about `git add` type

```
git help add
```

### 2.3.1 Simple work cycle

Suppose we want to make some code changes in a file `foo.r` that already exists in the master repo. A typical work sequence would be as follows.

1. Change directory (`cd`) to the local repo.
2. If collaborating with others, you might want to make sure that you local repo is up to date with the master copy on `github`, using `git pull` to get ('pull') any changes from `github`. (`git diff @{upstream}` can be used to view the changes first, if you prefer.)
3. Work on `foo.r` until you are happy with the changes made, and have tested them.
4. Have `git` take a snapshot of the changes made using something like

   ```
   git commit -a -m"added a wibbleblaster to foo.r"
   ```

   The `-a` option tells `git` to take a snapshot of every file that it is tracking that you have changed. If you omit `-a` then you must tell `git` which files you want it to snapshot explicitly, for example using `git add foo.r`, before committing. The option `-m` adds a message describing the changes. If you omit the `-m` then `git` will open an editor, in which you enter the message - this can be useful if you want to include a longer set of comments. These comments can be viewed on the `github` repo, providing a useful record of what the individual code changes were for.
5. Now 'push' the changes to `github`.

```
git push
```

If only you are working on the code, then you will be done at this point, but if others are working on it at the same time, then the `push` command may fail because your new code conflicts with the code someone else has pushed to `github` since you last pulled the repo. How to fix this is described next.

Note that `git log` lets you view the commit history of your project (to review what has been done and why).

### 2.3.2 Simple conflict resolution

If your `git push` command fails (and it will tell you if it has!) then you need to resolve the conflict. This is not so hard.

1. After your `git push` command has failed, issue the command

   ```
   git pull
   ```

   This will get the latest versions of the files from the `github` repo, compare them to your local copies, and attempt to resolve conflicts automatically where possible. Where auto-fixing is not possible, it will modify your local copies so that whenever there is a conflict your local files contain both your code and the conflicting code from the `github` master, with clear marking of which is which.

2. You check the conflicting files, changing the code to resolve any remaining flagged conflicts (often just selecting one or other of the alternatives).

3. Now redo the commit and push steps

   ```
   git commit -a -m"resolved conflict in favour of small end cracking"
   git push
   ```

   If this fails because a teammate has meanwhile made another change, you really need to sort out your team communication - `git` can't do that for you.

### 2.3.3 Adding and deleting files with `git`

You can add as many files as you like to your local copy of the repo, but `git` will take no notice of them until you tell it to. For example, suppose you created a file `bar.r` which should be treated as part of the project, tracked and included in the master repo on `qithub`

```
git add bar.r
git commit -a -m"added file bar.r containing the gribbler code"
git push
```

(You could omit the `-a` in this case, as the preceding `add` command will ensure `bar.r` is included in the next commit.)

You might also want to remove files, of course.

```
git rm bar.r
```

deletes `bar.r` from the local copy, and will cause it to be removed from the master repo at the next `commit` and `push`.

## 2.4 More advanced use

The above covers the most basic use of `git`. It is sufficient for working on small projects in small teams on this course, and for understanding the basic principles of version control systems. We have not covered some key components of `git` and `github` that are extremely useful for larger projects. In particular we have not covered project *branches*. `git` allows you to simultaneously have several versions of your project (*branches*) in addition to the master version. These versions can all be tracked and backed up, just as the main master branch is.

A typical use of branches is to code and work on complicated modifications - tracking and backing up the changes in those modifications, while not modifying the main project until is is clear that the modified code is really working and ready to be merged into the main project.

To give a concrete example, suppose you maintain a large R package, which relies on code written in C and Fortran, and you decide that it would be advantageous to replace all the Fortran code with C code. This is a major undertaking, and you would not want to simply start work on the stable working code for the main package, since this will almost certainly break it initially. That would be a real nuisance if you then needed to deal with a minor bug in the original package, but had only the half completed unstable modified code available. Much better to create a branch of the project on which to develop the new C based code, only merging into the master branch, once everything in the revision is fully working.

### 2.5 A simple `git`/`github` exercise

At this stage, it is *essential* that you try out `git` and `github`. So before reading on try the following exercise. If you are not sure how to do any part, then refer back to the material above that you have just read.

1. Using a web browser, set yourself up a new repo on `github`, and edit the default `README.md` file on `github` to say something interesting (making sure to save and commit the change).
2. Clone your `github` repo to your local machine.
3. Add a file to your local repo, edit it and add it to the files tracked by `git`.
4. Commit your edits and push the repo to `github`.
5. Check that the repo on `github` now contains your newly added file.
6. As soon as you know someone else on the course to work with, try out sharing repositories, and resolving conflicts.

## 3 Programming for statistical data analysis

Programming is the process of writing instructions to make a computer perform some task. The instructions have to be written in standard way that both the programmer and the computer can interpret. The rules and key words defining such a standard way of communicating define a computer language. There are many alternative languages designed for different classes of task. Here we will concentrate on the R language (and environment) for programming with data, which is widely used for the statistical analysis of data. A huge amount of statistical analysis software is written in R and is freely available.

*Statistical* analysis of data is concerned with the analysis of data that are in some sense a random *sample* from a larger *population*. We want to learn about the population from the sample, without being misled by particular features of the sample that arose by chance as part of the random sampling process. The random sampling approach is powerful because:

1. it allows reliable conclusions with known levels of accuracy to be drawn without having to gather data from the whole population, which may be impossible, or prohibitively expensive.
2. random sampling eliminates the unknowably large bias that occurs if we try to learn about the population from a non-random sample, replacing that bias with a random uncertainty of known magnitude.

Note that while 'population' might mean something concrete like 'population of people in the UK' it might be much more abstract like 'the population of all experimental results that this experiment could have produced when replicated under the same conditions'. Random Variable

To understand the difference between statistical and non-statistical data analysis consider data on reported cases of Covid-19 each day. Many charts of these data were produced, and analyses were performed, such as producing running averages or looking for trends in the data. None of these analyses are statistical, as no attempt was made to consider what population the case data might be viewed as a random sample of. They are certainly not a random

检测本身并不具备严格的随机性，同时存在很多系统性的因素导致样本选择偏差。

sample of the people who had Covid-19 on a given day, and neither is it remotely clear how the number of cases relates to the number of people with Covid on a particular day. The government and media presented these data *as if* they were a random sample from the population of Covid cases, but this was simply misleading. Any decent applied statistician should be able to list several sources of bias likely to occur by treating them as such.

In contrast, each week the UK Office for National Statistics published the results of testing a randomly selected sample of the UK population for Covid. The analysis included estimates of trends and uncertainties, from a proper statistical analysis. Unsurprisingly the ONS analysis often appeared to contradict the naive interpretations of the case data given by the media and government. More surprisingly the media and government seemed to give more weight to the non-statistical analyses of case data than to the statistical analyses of the ONS data, despite the latter being essentially unbiased and of known accuracy, while the former had bias of unknown magnitude. Even more surprisingly, the UK appears to have been unique in even conducting unbiased sampling to establish Covid prevalence.

The Covid cases example illustrates why statistical analysis software is dangerous. The easier software is to use, the easier it is to produce an analysis of data that resembles a statistical analysis in every superficial respect, but not in the key one that the data are in some sense a random sample from a population of interest. Beware!

幸存者偏差

### Exercise

The following link is to an article in a national newspaper by two professors of statistics that appeared on 19th April 2020. `https://www.theguardian.com/commentisfree/2020/apr/19/coronavirus-deaths-data-uk` There is an astonishing statement in the second paragraph. Identify the statement and what is wrong with it.

# 4 Getting started with R

In this course we will concentrate on learning programming skills that as far as possible are transferable to other programming languages in addition to R. For this reason we will largely stick to programming using what is available with R itself, and we will avoid over-reliance on any particular set of add on packages. There is an add on package providing functions for almost any simple task you might want to accomplish in R. The fact that we will here examine how to programme tasks for which it would be simpler to find an add on package, is not through a desire to re-invent the wheel. The point is not to learn the quickest way to perform the particular task, but rather to illustrate how to programme.

A word of warning. In much university work, getting something 80% right is a first class performance. Unfortunately programming is not like that. 80% right means 20% wrong, and 20% wrong computer code will likely result in your programme doing 0% of what it is supposed to do. This fact calls for a more careful approach to working than is necessary for other topics.

## 4.1 A first R session

To get started, let's do some trivial things in R. Start `R` or `Rstudio`, go to the terminal window and type

```
a <- 2
```

You just created an object `a` that contains the number 2. `<-` is the assignment operator. If you assign something to an object that does not yet exist, it is created. If you type the name of an object at the R terminal and nothing else, then the contents of the object gets printed. Exactly how this is done depends on the class of the object - more later. For example (> is just the R prompt, not something to type):

```
> a
[1] 2
```

We can make objects from other objects of course. For example

```
> b <- 1/a
> b
[1] 0.5
```

R is a functional programming language: it is structured around functions that take objects as arguments and produce other objects as results. Many functions are built in to base R and even basic operators like + and / are actually implemented as functions. Suppose we want to create a function to take arguments $x$ and $y$ and return the value of $x\cos(y-k)$ where $k$ is a constant usually taking the value 0.5. Here is the code to define such a function object, named `foo` in this case.

```
foo <- function(x,y,k=0.5) {
  x * cos(y - k)
}
```

Curly brackets, `{}`, enclose the R code defining how the function arguments are turned into its result. There can be as many lines of code as you like, but what ever is produced on the last line is taken as the object to be returned as the function result. `k=0.5` is used to indicate that if no value of `k` is supplied, then it should take the default value of 0.5. This default system is used extensively by R and its add on packages. Let's try out `foo`.

```
> foo(3,2) ## using default k=0.5
[1] 0.2122116
> foo(3,2,0) ## setting k=0
[1] -1.248441
```

R is an interpreted language. Code[2] is interpreted and executed line by line as it is encountered. This contrasts with compiled languages, where code is converted to binary instructions en masse, and this binary 'machine code' (the native language of the computer's processor) is then executed separately. R evaluates lines of code when they appear complete, and a line end has been encountered. If you want to split code over several lines then you need to be careful that the line does not appear complete before you meant it to be. e.g. suppose we want to evaluate $a = 1+2+3+4+5$

```
> a <- 1 + 2 + 3 ## split line but it looks complete to R!
> + 4 + 5
[1] 9
> a
[1] 6 ## oops
> a <- 1 + 2 + 3 + ## split line that does not look complete
+ 4 + 5
> a
[1] 15 ## success
```

Several complete statements can be included on the same line by separating them with a '`;`'. e.g.

```
a <- 2; b <- 1/a; d <- log(b)
```

Now leave R with the `q()` command. By default you will be asked if you want to save your workspace - usually you do not. You can avoid being asked by typing `q("no")`.

## 4.2 Dissecting a simple programming example

Let us continue with an example of a simple data manipulation task in R. Suppose that we have a vector of 1 or 2 digit numbers and want to create a new vector of the individual digits, in order. For example if the original vector is $[12, 5, 23, 2]$ the new vector should be $[1, 2, 5, 2, 3, 2]$. Before trying this we need to write down *how* we are going to do it. For example:

1. Identify the number and locations of the double digit numbers in the original vector (and hence the length of the new vector).
2. Work out the locations of the 'tens' digits in the new vector, compute and insert them.
3. Work out the locations of the 'units' digits in the new vector and insert them.

[2] I'll use 'code' to mean programmes and commands written in R (or indeed other computer languages).

The following is R code for one way of implementing this. It could be typed, line by line, into the R console, but it is better to type it into a file and then copy and paste to the R console, or `source` the file into R, or run it in Rstudio.

```
x <- c(10,2,7,89,43,1) ## an example vector to try out
ii <- which(x%/%10 > 0) ## indices of the double digits in x?
xs <- rep(0,length(ii)+length(x)) ## vector to store the single digits
iis <- ii+1:length(ii)-1 ## where should 10s digits go in xs?
xs[iis] <- x[ii]%/%10 ## insert 10s digits
xs[-iis] <- x%%10 ## insert the rest (units)
```

Anything after `#` on a line is ignored by R, so is a comment. It is a good idea to use lots of these. Here is what the code does in detail, line by line.

1. `x <-c(10,2,7,89,43,1)` creates an example vector to work on. The `c` function takes the individual numbers supplied to it, and concatenates them into a single vector (it can also concatenate vectors). The results are stored in vector `x` using the assignment operator `<-`. Notice how `x` is created automatically by this assignment. Unlike in many computer languages, we do not have to declare `x` first.

2. `ii <-which(x%/%10 > 0)` creates a vector, `ii`, of the indices of the double digit numbers in `x`. It does this by computing the result of integer division by 10, for each element of `x`, using the `%/%` operator, and then testing whether this result is greater than 0. The result of `x%/%10 > 0` will be a vector of `TRUE` or `FALSE` values of the same length as `x`. For the given example it is (`TRUE`, `FALSE`, `FALSE`, `TRUE`, `TRUE`, `FALSE`). This vector is supplied directly to the `which` function, which returns the indices for which the vector is `TRUE` (1,4 and 5 for the example given).

3. `xs <-rep(0,length(ii)+length(x))` creates a vector of zeroes into which the individual digits will be inserted, using the `rep` function. `rep` simply repeats its first argument the number of times specified in its second argument. The `length` function returns the number of elements in an existing vector. We need `xs` to be the length of `x` plus an extra element for each double digit number.

4. `iis <-ii+1:length(ii)-1` creates a vector, `iis`, containing the indices (locations) of the 10s digits in `xs`. We have to account for the fact that each time we insert a digit, all the digits after move along one place, relative to where they were. So, the first 10s digit will occupy the same slot in `x` and `xs`, but the next 10s digit will be one element later in `xs` than in `x`, the next 2 elements later, and so on. `1:length(ii)-1` creates a sequence 0,1,2... to add to `ii` to achieve this. It uses the `:` operator — if `a` and `b` are integers (and $a \leq b$) `a:b` generates the sequence $a, a+1, a+2, \ldots, b$.

   先create sequence 再 -1

5. `xs[iis] <-x[ii]%/%10` computes the 10s digits of the double digit numbers, indexed by `ii`, using `x[ii]%/%10`, and assigns them to the elements of `xs` indexed by `iis`.

6. `xs[-iis] <-x%%10` computes the units digit for all the numbers in `x` using `x%%10` where `%%` is the operator computing the remainder after integer division. These digits are stored in the elements of `xs` *not* reserved for 10s digits. That is, `xs[-iis]`, all the elements *except* those indexed by `iis`.

   -iis：表示除 iis 以外的位置

Make sure you really understand what this code is doing. Run it one line at a time in `R`, and examine the result created by each line to make sure you understand exactly what is happening. To see what is in an R object, just type its name at the console, and it will be printed, by default. For example:

```
> xs ## > just denotes the R prompt here
[1] 1 0 2 7 8 9 4 3 1
```

The preceding example has quite a bit of R packed in, and we used quite a few R functions and operators on the way. R provides a large number of functions for a variety of tasks, and the available add on packages vastly more. You can't hope to learn them all, so it is essential to learn how to use the R help system. This is easy. For any operator of function you know the name of then just type `?` followed by the function name, or the operator in quotes. For example

```
?which ## get the help for the 'which' function
?"<-" ## get help for the assignment operator
help.start() ## launch html help in a browser
```

. . . the last option is often best if you are not sure what you are looking for.

## 4.3 A second simple example: data are not always numbers

R vectors are not restricted to containing numbers. Character strings are another common data type that we can hold in a vector. For example `x <-c("jane","bill","sue")` creates a 3-vector, containing the 3 given character strings. Functions are provided for manipulating such character string data in various ways. For the moment suppose we want to achieve the same task as in the previous example, but for the case in which the numbers are supplied as character strings, and we want the separated digits as character strings too[3].

We can use more or less the same approach as in the last section, but with one slight modification to the logic. For numbers it was easy to separate out 10s and units, with only the 2 digit numbers having a 10s digit. When the numbers are represented as character strings it is easier to separate out first and second digits, with only the two digit numbers having second digits. This means that rather than finding the indices of 10s digits, we'll find the indices of second digits. Here is the modified code:

```
x <- c("10","2","7","89","43","1") ## example vector
ii <- which(nchar(x)>1) ## which elements of x are double digit?
xs <- rep("",length(ii)+length(x)) ## vector to store the single digits
iis <- ii+1:length(ii) ## where should second digit go in xs?
xs[iis] <- substr(x[ii],2,2) ## insert 2nd digits
xs[-iis] <- substr(x,1,1) ## insert 1st digits
```

Line by line, here is what it does:

1. `x <-c("10","2","7","89","43","1")` example vector, as before, but now of character type.
2. `ii <-which(nchar(x)>1)` uses the `nchar` function to count the characters in each element of `x`. `which(nchar(x)>1)` returns the indices for which the corresponding element of `x` has $> 1$ character.
3. `xs <-rep("",length(ii)+length(x))` creates `xs` as before, but this time it is a character vector.
4. `iis <-ii+1:length(ii)` computes the locations for second digits in `xs`. Same idea as before, but second digits are all located one place after the 10s digits.
5. `xs[iis] <-substr(x[ii],2,2)`. Function `substr` is used to obtain the 2nd character from each 2 digit element of `x` (`ii` indexes the 2 digit elements). `substr(x[ii],2,2)` extracts the characters between characters 2 and 2, from the elements of `x[ii]` and returns them in a vector, which is copied into the appropriate locations in `xs`.
6. `xs[-iis] <-substr(x,1,1)` the first digits are then inserted in the locations not reserved for second digits.

### 4.3.1 Another simple text processing task

The above example is a bit artificial. Let's consider a slightly more realistic task to undertake on character data. Suppose we have a string containing some 'poetry', and we want to count the number of words, tabulate the number of letters per word, count the number of words containing at least one 'e' and mark all words containing an 'a' and an 'e' with a '*'. Here is R code to do this.

```
poem <- paste("Inside me is a skeleton, of this I have no doubt,",
      "now it's got my flesh on, but it's waiting to get out.")
pow <- strsplit(poem," ")[[1]] ## vector of poem words
```

strsplit() 返回一个列表，[[1]] 取第一个元素

[3] As the point is to provide illustration of string handling, I'll resist the temptation to use `as.numeric` to convert the character data to numbers, run the previous code, and then use `as.character` to convert back.

```
n.words <- length(pow) ## number of words
freq <- tabulate(nchar(pow)) ## count frequency of n-letter words
ie <- grep("e",pow,fixed=TRUE) ## find 'e' words
n.e <- length(ie) ## number of 'e' words
ia <- grep("a",pow,fixed=TRUE) ## find 'a' words
iea <- ia[ia %in% ie] ## find words with 'e' and 'a'
pow[iea] <- paste(pow[iea],"*",sep="") ## mark 'e' 'a' words
paste(pow,collapse=" ") ## and put words back in one string.
```

e出现的位置组成的向量

grep() : 匹配字符　　fixed = TRUE 不使用正则表达式

%in% 返回一个 TRUE 和 FALSE 的向量

默认 seq = " "

Line by line it works like this:

1. The first line just creates a text string, `poem`, containing the given text. The `paste` function joins the two given strings into one string. The only reason to use it here was to split the string nicely across lines for these notes - we could just as well have written everything in one string to start with.

2. `pow <-strsplit(poem,"")[[1]]` splits `poem` into a vector of its individual words, using `strsplit(poem,"")`, which splits the string in `poem` at the breaks given by spaces, `" "`. `strsplit` can take a vector of strings as its first argument, and returns a *list* of vectors containing the split strings. In our case the list only has one element, which is what the `[[1]]` part of the code accesses.

3. `n.words <-length(pow)` counts the words in `poem`, since there is one element of `pow` per word.

4. `freq <-tabulate(nchar(pow))` counts how many 1 letter, 2 letter, 3 letter, etc. words are in `poem`. First function `nchar` counts the letters in each word and then `tabulate` tallies them up. Really we should have stripped out punctuation marks first - `gsub` could be used to do this.

5. `ie <-grep("e",pow,fixed=TRUE)` finds the indices of the words containing an 'e' using the `grep` function. By default `grep` can do much more complicated pattern matching using 'regular expressions' (see `?regex`). For the moment this is turned off using `fixed=TRUE` (otherwise characters like `.` and `*` are not matched as you might expect, but treated differently).

6. `n.e <-length(ie)` is the count of 'e' words.

7. `ia <-grep("a",pow,fixed=TRUE)` finds the indices of the words containing an 'a'.

8. `iea <-ia[ia %in% ie]` finds the indices of words containing an 'a' and an 'e'. `ia %in% ie` gives a `TRUE` for each element of `ia` that occurs in `ie` and a `FALSE` for each element of `ia` that doesn't. Hence `iea` will contain the indices of the words containing both letters.

9. `pow[iea] <-paste(pow[iea],"*",sep="")` adds a '*' to each word in `pow` containing an 'e' an an 'a'. The `paste` function is used for this, with `sep=""` indicating that no space is wanted between a word and '*'.

10. `paste(pow,collapse="")` is finally used to put the words in `pow` back into a single string. It is the setting of `collapse` to something non-NULL (here a space) that signals to `paste` that this should happen.

## Exercises

1. Look up the help page for `gsub` function, and use `gsub` to remove the commas and full stops from the poem before tabulating the length of words.

```
> poe <- gsub("[,.]", "", poem)
```

2. For the original `poem`, write R code to insert a line end character, `"\n"`, after each comma or full stop, and print out the result using the `cat` function.

```
> poen <- gsub("[,.] ", "\n", poem)
> cat(poen)
```

3. Run the code `set.seed(0);y <-rt(100,df=4)`. Use the `hist` function to visualize `y`. Now using the `mean` and `sd` functions to find the mean and standard deviation of `y`, write code to remove any points in `y` that are further than 2 standard deviations from the mean, and calculate the mean of the remainder.

4. Using your code from the previous section to write a function that will take any vector `y` and compute its mean after removal of points more than `k` standard deviations from the original mean of `y`. Set the default value of `k` to 2.

```
trm <- function(y, k = 2){
  y_mean <- mean(y)
  y_sd <- sd(y)

  y_filter <- y[abs(y) <= y_mean + k*y_sd]

  y_filter_mean <- mean(y_filter)
  y_filter_sd <- sd(y_filter)

  print(paste("After removal of points more than k standard deviations,
      mean = ", y_filter_mean, " sd = ", y_filter_sd))
}
```

# 5 A slightly more systematic look at R

When you start the `R` programme, two important things are created. The first is an `R` terminal, into which you can type commands written in the R programming language - this is visible. The second is an *environment*, known as the user workspace or global environment which will hold the objects created by your commands - this is invisible, but is there as an extendable piece of computer memory. In R an environment consists of a set of symbols used as the names of objects along with the data defining those objects (known together as a *frame*) and a pointer to an enclosing 'parent' environment. R makes extensive use of sets of nested environments, but we don't need to go into too much detail on this aspect at the moment.

Like any computer language, the R language defines basic data structures, key words used to control programme flow, operators and functions, plus a set of rules about how these things are used and combined. This section introduces these.

## 5.1 Objects, classes and attributes

Everything in R is an object living in an environment, including R commands themselves. Objects have *classes* which R can use to determine how the object should be handled (for example which version of the print function is appropriate for it). Objects can also be given *attributes*: these are basically other objects that have been 'stuck onto' the object and are carried around with it. A bit like a set of virtual post-it notes. Attributes are useful for storing information about an object. For example, matrices in R have class `"matrix"` and a `dim` attribute. The `dim` attribute stores the number of rows and columns in the matrix.

## 5.2 Data structures: vectors, arrays, lists and data frames

Let's look at the basic data structures you *must* know about to programme in R. When reading through this, try out the code yourself in R, and also try modifying it to check your understanding.

### 5.2.1 Vectors and recycling

The most basic type of data structure in R is a vector (a one dimensional array). `x[i]` accesses element `i` of vector `x` (`i` is a positive integer). Even scalars are just vectors of length 1 (so e.g. `3[1]` is perfectly valid and evaluates to 3, of course). As we have already seen, vectors can store data of different *types*: integer or real numbers (type 'double'), character strings, logical variables. The class of vectors is simply determined by the type of thing they contain. Here are some basic examples.

```
> a3d <- c(TRUE,FALSE,FALSE) ## create an example logical vector
> class(a3d) ## its class
[1] "logical"
> typeof(a3d) ## its type
[1] "logical"
> a3d[2:3]   ## print its 2nd and 3rd elements
[1] FALSE FALSE
> a3d[2] <- TRUE ## change the 2nd element
> a3d        ## print the resulting vector
[1] TRUE TRUE FALSE
>
> bb <- 1:10 ## create numeric example
> class(bb) ## check class
[1] "integer"
> bb[c(1,4,9)] <- c(.1,-3,2.2) ## change selected elements
> class(bb) ## note automatic change of class
[1] "numeric"
> typeof(bb) ## how it is actually being stored
[1] "double"
> bb[3:8]  ## print elements 3:8
[1] 3 -3 5 6 7 8
```

Whatever the type of a vector, some of its elements can always be set to `NA` (not available), if required. Also numbers can take the values `Inf`, `-Inf` and `NaN` (not a number e.g. $\log(-1)$).

Since vectors are the basic data structure, operators and most functions are also vectorized - they operate on whole vectors. For example given two vectors, `a` and `b` of the same length then `c <-sin(a) *b` actually forms `c[i] <-sin(a[i]) *b[i]` for all `i` from 1 to the length of the vector. Similarly `c <-a *b + 2` actually forms `c[i] <-a[i] *b[i] + 2` for the same set of `i` values.

Notice how the scalar `2` got reused for each `i` in that last example. So is a scalar more than just a length one vector after all? Actually no, `2` is being treated like any other vector — R has a **recycling rule** for vector arithmetic. Any operator that combines two vectors will recycle the values in the shorter vector as many times as required to match the length of the longer vector. So if a vector contains only one value, that one value is just recycled as often as needed. Here are a couple of examples

短向量循环使用

```
> a <- 1:4 # a 4-vector  1 2 3 4
> b <- 5:6 # a 2-vector  5 6
> a*b    # multiplication with recycling
[1] 5 12 15 24
> b <- 5:7 # a 3 vector
> a*b    # multiplication with recycling
[1] 5 12 21 20
Warning message:
In a * b : longer object length is not a multiple of shorter object length
```

Vectors can be accessed and subsetted using logical vectors in place of indices. The vector should be the same length as the vector being accessed, and will be recycled if not. The elements of vectors can also be named, and can be accessed by name as well. Here are some illustrations (obviously names are not necessary for logical access):

```
> x <- 1:5; names(x) <- c("fred","sue","bill","eve","bob") ## named vector
> x
fred sue bill eve bob
   1   2    3   4   5
> x[c(TRUE,FALSE,TRUE,FALSE,FALSE)] ## subset using logical indexing vector
fred bill
   1    3
> x[c(TRUE,FALSE)] ## ... logical indexing vector recycled!
fred bill bob
   1    3   5
> x[x>3] ## common case where logical vector generated in situ by a condition
eve bob
  4   5
> x[c("sue","eve")] ## access by name
sue eve
  2   4
```

### 5.2.2 Matrices and arrays

While vectors are one dimensional arrays data, matrices are 2 dimensional arrays, and in general an array can have as many dimensions as we find useful. We can create an array with the `array function`. For example:

```
a <- array(1:24,c(3,2,4))
```

creates a $3 \times 2 \times 4$ array, filling it with the numbers given in the first argument. To access the array we just give the dimension indices for the elements required. Leaving an index blank implies that we require all elements for that dimension. For example.

```
> a[3,2,3] ## element 3,2,3
[1] 18
> a[1:2,1,]
     [,1] [,2] [,3] [,4]
```

```
[1,]    1    7   13   19
[2,]    2    8   14   20
```

notice how the second example accesses all elements for which the first index is 1 or 2, and the second index is 1.

Arrays are actually stored as vectors, with class `"array"` and a `dim` attribute. The `dim` attribute is a vector containing the length of each dimension (so $3, 2, 4$ above). Since the underlying storage is vector, we can also access it as such, we just need to know that the data are stored in the vector in 'column major order'. For example if $d$ is the `dim` attribute of a 3 dimensional array, $a$, then $a[i, j, k]$ is equivalent to $a[i + (j-1)d_1 + (k-1)d_1d_2]$. For example

列主序：先从第一个维度开始填

```
> d <- dim(a) ## get the 'dim' attribute
> a[3+1*d[1]+2*d[1]*d[2]] ## vector access to a[3,2,3]
[1] 18
```

Notice that this is quite useful if you need to fill in or access several scattered individual elements of an array at once.

Two dimensional arrays, **matrices**, play a central role in statistics. Data properly arranged for analysis are usually in matrix form with columns as variables and rows as observations (often referred to as 'tidy data'), while many statistical methods rely heavily on matrix computations. Hence matrices are treated as a special class of array, with their own `"matrix"` class, a `matrix` function used to create them, special operators for matrix multiplication and other matrix products, and functions implementing many matrix decomposition and matrix equation solving tasks. We will cover this in more detail later, but here is a quick example, which uses the matrix multiplication operator, `%*%`.

```
B <- matrix(1:6,2,3); B ## create a matrix (filled by col)
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> B[1,2] <- -1             ## change element 1,2
> a <- c(.3,-1.2,2.3)     ## a 3-vector
> B %*% a                  ## matrix multiplication
     [,1]
[1,] 13.0
[2,]  9.6
> B*a                      ## element wise multiplication with recycling!!
     [,1] [,2] [,3]
[1,]  0.3 -2.3 -6.0
[2,] -2.4  1.2 13.8
```

Do make sure you *really* understand the difference between the last two commands! It *really* matters.

### 5.2.3 Lists

Lists are the basic building blocks of all sorts of complicated objects in R. Each item of a list can be any sort of R object, including another list, and each item can be named. Suppose `a` is a list. Individual items can be accessed by number using `a[[i]]` where `i` is an integer. If an item has a name, `"foo"` for example, it can also be retrieved that way using `a[["foo"]]` or `a$foo`. We can also access sublists, also by name or number. e.g. `a[c(2,4)]` or `a[c("foo","bar")]` both produce 2 item lists. Note the difference: `[[]]` retrieves the item, `[]` retrieves a list of the required items (even if there is only one). Here is a simple example, which uses function `list` to create an example list.

```
> stuff <- list(a=1:6,txt="furin cleavage site",l2 = function(x) log(x^2),
+       more = list(a="er",b=42))
> stuff[[1]]
[1] 1 2 3 4 5 6
> stuff[["l2"]]
function(x) log(x^2)
```

[ ]：用来截取子列表
[[ ]]：用来获取指定位置的元素

```
> stuff$a
[1] 1 2 3 4 5 6
> stuff[c(1,2)]
$a
[1] 1 2 3 4 5 6
$txt
[1] "furin cleavage site"
> stuff$g <- runif(10) ## add an element
> names(stuff)
[1] "a" "txt" "l2" "more" "g"
> names(stuff)[1:2] <- c("b","text") ## change some names
```

### 5.2.4 Data frames, factors and dates – statistical data structures

R provides two types of data structure that are particularly useful for statistics: **factor** variables are a special class of variables especially useful for data that consist of labels that serve to classify other data; **data frames** are 2 dimensional arrays of data where the columns are variables, *which can be of different types*, and the rows are observations.

**Factors** are vectors of labels. For example a clinical trial might record the `sex`, `nationality` and `treatment` received for each subject. All three are labels that categorize the subject. The different values that the label can take are known as *levels* of the factor (although they usually have no ordering, and it they do it is generally ignored). For example in a multi-centre vaccine trial `nationality` might have levels `"GB"`, `"Brazil"`, `"USA"`. In R factors are of class `"factor"` and the `levels` function can be used to access their `"levels"` attribute. In fact the actual labels of a factor variable are only stored in the `"levels"` attribute, the variable itself is a set of integers indexing the levels. Why all this fuss? Because factor variables are very useful in statistical modelling, and setting them up this way makes them easy for modelling functions to handle. Here is a simple example.

factor 中实际存储一个个的整数，对应 levels 中标签的索引

```
> fac <- factor(c("fred","sue","sue","bill","fred"))
> class(fac)
[1] "factor"
> fac      ## default printing for class factor
[1] fred sue sue bill fred
Levels: bill fred sue
> levels(fac) ## extract the levels attribute
[1] "bill" "fred" "sue"
> as.numeric(fac) ## look at the underlying coding
[1] 2 3 3 1 2
```

**Data Frames** are basically matrices, in which the columns have names and can have different types (numeric, logical, factor, character, etc). They can be accessed like matrices, or like lists. They provide the best way of organising data for many types of statistical analysis. Here is a basic example

二维表格

```
> dat <- data.frame(y = c(.3,.7,1.2),x = 1:3,fac = factor(c("a","b","a")))
> dat    ## a data.frame
   y x fac
1 0.3 1   a
2 0.7 2   b
3 1.2 3   a
> dim(dat) ## like a matrix
[1] 3 3
> dat$fac ## and like a list!
[1] a b a
Levels: a b
```

**Dates** can conveniently be represented using objects of class `"Date"`. See `?Date` for a description, `?as.Date` and `format.Date` for converting between text and `Date` format and `?julian` for converting to 'time since some origin', for example. Times and dates can also be stored together in objects of class `"POSIXct"`. See

`?DateTimeClasses` for details. Particular advantages of these representations are the ability to reliably compute time differences, and to extract parts of the time and date. Here is a short example, sufficient to give an idea of what is possible.

```
> ## create date-time data (format is actually default here)
> d <- as.POSIXct(c("2016-06-28 21:30:00","2016-06-29 22:00:00",
+                   "2016-06-30 22:30:00"),format="%Y-%m-%d %H:%M:%S")
> julian(d,origin=as.Date("2016-01-01")) ## get day since Jan 1 2016
Time differences in days
[1] 179.8542 180.8750 181.8958
attr(,"origin")
[1] "2016-01-01 GMT"
> format(d,"%T") ## extract times as text
[1] "21:30:00" "22:00:00" "22:30:00"
> as.Date(d)     ## get dates as class 'Date'
[1] "2016-06-28" "2016-06-29" "2016-06-30"
```

Date : **日期（天数）**
POSIXct : **日期** + 时间 **（秒数）**
julian() : 计算时间差
format() : 转换为文本格式

## Exercises

1. Run the code

   ```
   set.seed(5); n <- 2000
   w <- runif(n)
   A <- matrix(runif(n*n),n,n)
   system.time(B <- diag(w) %*% A )
   ```

   B <- w * A

   **将 A 当作一个向量与 w 相乘，**w 的长度刚好对应 A 每一列的长度，循环利用 w，则 w **的第** i 个元素刚好对应 A **的第** i **行。**

   noting how long the last line takes to run. `diag(w)` forms the diagonal matrix with leading diagonal elements given by the elements of `w` (try an $n = 5$ example if that's not clear). By considering what actually happens when a diagonal matrix multiplies another matrix, and how matrices are stored in R, find a way of calculating `B` using the recycling rule without using `diag` or `%*%`. Time this new code, and check that it gives the same answer as the old code (the `range` function might be useful).

2. Write a single line of code to compute what number day of the year it is today.

   julian(as.Date("2024-09-23"), origin = as.Date("2023-12-31"))

3. Given the importance of factor variables in statistical modelling, it is worth some effort in understanding exactly how they are represented, as some basic tasks with factors are much easier if you understand this. First set up a simulated factor variable:

   ```
   a <- factor(sample(c("fred","george","sue","ann"),20,replace=TRUE)); a
   ```

   You'll see that the levels of the factor have been ordered alphabetically. Since factors are just groups, and don't usually have a natural order that may be fine. However, plot functions, for example, will often use the level order to decide plotting order, and you might want to change that. Similarly statistical models often treat the first level differently from the others, so again you may want to control which it is. If you want the levels to be ordered differently then you can supply an order like this:

   ```
   b <- factor(a, levels = c("ann","sue","fred","george")); b
   ```

   Examine the underlying numeric values stored in `a` and `b` to make sure you understand exactly what has happened here. Finally, extract the levels of `a` to a vector `al` using the `levels` function, and the numeric representation of `a` to another vector, `an`. Now use just `al` and `an` to reproduce a vector of the level labels for each entry in `a`.

   > an <- as.numeric(a); al <- levels(a); av <- al[an]; av

4. Create a list containing: a $3 \times 3$ matrix `M` of $U(0, 1)$ random numbers (see `?runif`); a list `b` itself containing a string `"foo bar"` and a vector containing `5:10`; a vector `v` containing `-3, 5, -1`. From your first list create a second list containing only `M` and `v`, using a single line of code. Delete `v` from the first list, and change the item names to `A` and `blist`.

M <- matrix(runif(9), nrow = 3, ncol = 3)

b <- list("foo bar", 5:10)

v <- c(-3, 5, -1)

first_list <- list(M = M, b = b, v = v)

second_list <- first_list[c("M", "v")]

first_list$v <- NULL

names(first_list) <- c("A", "blist")

## 5.3 Attributes

As already mentioned, attributes are simply R objects attached to other objects, and carried around with them (a bit like a set of 'post it' notes). They can be useful for storing things that are somehow properties of an object. We can find out about an object's attributes using the `attributes` function. For example

```
> attributes(PlantGrowth) ## PlantGrowth is an R data set
$names
[1] "weight" "group"

$class
[1] "data.frame"

$row.names
 [1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
[26] 26 27 28 29 30

> A <- matrix(1:6,2,3) ## create a matrix
> attributes(A)  ## check its attributes
$dim
[1] 2 3
```

Single attributes can be queried, reset or created using the `attr` function. For example.

```
> attr(A,"dim") ## query attibute
[1] 2 3
> attr(A,"foo") <- list(1:3,"fred") ## create or reset attribute
```

Attributes can be quite useful when your programming task requires an object that modifies a standard object (e.g. a matrix) by adding some extra information to it, but you still want all the standard functions that work with the object to function as for the standard object (e.g. matrix decompositions).

### 5.3.1 str and object structure

Before moving on from data structures it is worth mentioning one useful function: `str` prints a summary of the structure of any R object. This is a good way of seeing what an object actually looks like, since many classes of object have their own `print` function, so that only rather limited information is printed when you simply type the object name at the command line. For example

```
> str(dat)      observations
'data.frame': 3 obs. of 3 variables:
 $ y : num 0.3 0.7 1.2
 $ x : int 1 2 3
 $ fac: Factor w/ 2 levels "a","b": 1 2 1
                with 2 levels
```

## 5.4 Operators

Here is a boring table of some operators in R.

| Operator | Description | Operator | Description | Operator | Description |
|---|---|---|---|---|---|
| `&` | logical AND | `\|` | logical OR | `<` | less than |
| `<=` | $\leq$ | `>` | greater than | `>=` | $\geq$ |
| `==` | testing equality | `!=` | not equal | `!` | logical negation |
| `<-` | assignment | `=` | assignment | `+` | addition |
| `-` | subtraction | `*` | multiplication | `/` | division |
| `^` | raise to power | `%/%` | integer division | `%%` | integer remainder (mod) |
| `%*%` | matrix multiplication | `%x%` | Kronecker product | `%in%` | logical match |

These operators act element-wise on vectors, except for the final row, which are matrix operators, or in the case of `%in%` operates on vectors of different lengths (see `?"%in%"`) In addition `||` and `&&` are logical OR and AND operators for use only with non-vector logical statements, typically for use in `if` or `while` statements.

For example suppose we want to set all elements of vector `x` to zero, for which $x[i]$ is between 1 and 2, or less than -2. `x[(x < 2 &x > 1) | x < -2] <-0` does it.

## 5.5 Loops and conditional execution

Many programming tasks require that some operation is repeated many times for different values of some index, or require that the choice of which operation to carry out should depend on some conditions.

Let's start with conditional execution of code. Often this is coded in an implicit vectorized way. For example `x[x<0] <-0` finds all the elements of `x` that are less than 0 and sets them to zero. i.e. we have set `x[i]` to zero only if originally `x[i]` was less than zero. But sometimes we need a more explicit way of doing this: `if` and `else` are used to achieve this. The basic form is

```
if (logical condition) {
  ## one version of code
} else {
  ## another version of code
}
```

Leaving out the `else` means that nothing is done if the `condition` is `FALSE`. For example we could simulate a coin toss:

```
if (runif(1)>.5) cat("heads\n") else cat("tails\n")
```

`runif` generates uniform random numbers on $(0, 1)$ and `cat` is a simple function for printing (`"\n"` produces a line end). In R you will often see code with the following sort of structure

```
a <- if (a<0) 0 else a + 1
```

i.e. what is assigned to an object depends on a logical condition. `if` statements can also be chained together of course. Here is a pointless example:

```
if (runif(1)>.5) {
  cat("heads\n")
} else if (runif(1)>.7) {
  cat("tails\n")
} else cat("also tails\n")
```

Let's move on to looping. The `for` loop is the most commonly used example. It repeats a set of R commands once for each element of some vector. The basic syntax is

```
for (a in vec) {
  ## some R code goes here
}
```

which repeats the code between the brackets[4], for `a` set to each value in `vec` in turn. Here's a simple example[5]

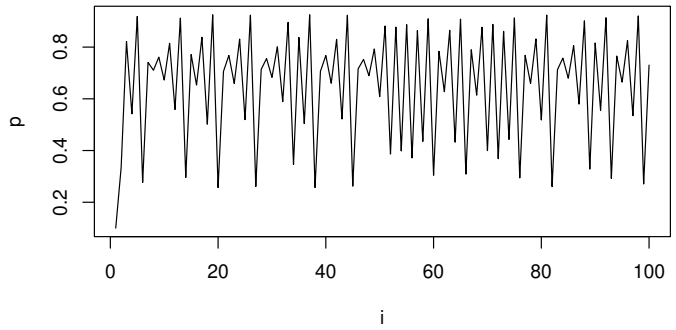
```
> vec <- c("I","am","bored")
> for (a in vec) cat(a," ")
I  am bored
```

Perhaps the most common use of a `for` loop is to loop over all integers between some limits. For example `for (i in 1:10) {...}` evaluates all the commands in `{...}` for $i = 1, 2, \ldots, 10$. (Take care if you have programmed in other languages – in R `for (i in 1:0) {...}` *will* execute the loop for 1 and then for 0). Here is an example iterating a chaotic map:

[4] you can drop the brackets if there is only one statement to execute at each iteration.

[5] for which a loop is in no way needed: you should be able to replace this with a single call to `cat`.

```
n <- 100;p <- rep(NA,n); p[1] <- 0.1
for (i in 2:n) p[i] <- 3.7 * p[i-1] * (1 - p[i-1])
plot(1:n,p,type="l",xlab="i")
```




What if I wanted to stop the above loop if `p[i]` exceeded 0.92? I could do that by using the `break` command inside the loop, to break out if the condition was met. i.e.

```
for (i in 2:n) {
  p[i] <- 3.7 * p[i-1] * (1 - p[i-1])
  if (p[i]>.92) break
}
```

Occasionally you might want to `repeat` indefinitely until a condition is met that causes you to `break`. The basic structure is this

```
repeat {
  some code
  if (condition) break
}
```

Another possibility is a `while` loop, something like

```
while (condition) {
  R code
}
```

but `for` loops are the most commonly used.

Note that for a vector oriented language like R, these sorts of tasks can often be accomplished efficiently by exploiting the fact that vectorized operations loop over vectors automatically, while vectors can also be subsetted so that we act only on the parts meeting some condition. Vectorized operations are usually much faster than explicitly coded loops, so generally it is a good idea to only write explicit loops when vectorization is not possible, or if the operations to be conducted at each loop iteration are expensive enough that the overhead of explicit looping is unimportant. For example, you should never write `for (i in 1:length(x)) y[i] <-x[i]` since `y <-x` does the same thing more efficiently, with less code:

```
> n <- 10000000; x <- runif(n) ## simulate 10 million random numbers
> system.time(y <- x) ## time a vector copy
   user system elapsed
  0.000 0.000 0.001
> system.time(for (i in 1:n) y[i] <- x[i]) ## time the equivalent copy loop
   user system elapsed
  0.538 0.028 0.567
```

In an interpreted language like R, there is more interpreting what to do than actual doing in the loop case.

## Exercise

1. Suppose you have vectors `x <-rnorm(20); z <-rnorm(20); y <-rnorm(20)`. Write (2 lines of) code to replace $x_i$ by $x_i^2$ if $z_i \neq 0$ and the following condition holds: $z_i$ is less than one or $y_i/z_i$ is negative.

2. Suppose you want to find the solution to the equation $f(x) = 0$ where $f(x) = \log(x) - 1/x$. A reasonable approach is bisection search. Start with an interval $[x_0, x_1]$ such that $f(x_0) < 0$ and $f(x_1) > 0$.[6] Defining $x_t = (x_1 + x_0)/2$, set $x_1 = x_t$ if $f(x_t) > 0$ and $x_0 = x_t$ otherwise. This process can be repeated until $f(x_t)$ is close enough to zero (or until $x_1 - x_0$ is small enough). Write code to find the solution to $f(x) = 0$ by bisection search, given that the solution (root) is somewhere in $[.1, 10]$ . Function `abs` finds the absolute value of a function, by the way.

3. Write code to create a $p \times p$ matrix and to fill it with zeroes below the leading diagonal, and ones on and above the leading diagonal, using loops.

4. Look up the `rep` function for creating sequences of numbers, try it out and then repeat question 2 using `rep` rather than looping. Use `system.time` to compare the speed of the code from questions 3 and 4 when $p = 1000$.

5. Now write code to efficiently zero all entries below the leading diagonal in a $p \times p$ matrix.

```
1.
for (i in 1:20) {
  if ((z[i] != 0) && (z[i] < 1 || y[i] / z[i] < 0)) x[i] <- x[i]^2
}
```

```
3.
mat <- matrix(0, nrow = p, ncol = p)

# 使用循环遍历矩阵的每个元素
for (i in 1:p) {
  for (j in i:p) {
    mat[i, j] <- 1
  }
}
```

```
2.
repeat {
   xt <- (x0 + x1) / 2

   if (f(xt) < 0) {
     x0 <- xt
   } else if (f(xt) > 0) {
     x1 <- xt
   }

   if (abs(f(xt)) < 0.01) {
     print(xt)
     break
   }
}
```

```
4.
system.time({
  # 这里放你用 `rep` 或循环实现的代码
})
```

```
5.
# 将主对角线以下的元素设置为 0
mat[row(mat) > col(mat)] <- 0
```

## 5.6 Functions

Functions were introduced in Section 4.1, but some more detail is required to write them effectively. Formally a function consists of an argument list, a body (the code defining what it does), and an environment (which is the environment where it was created). Generally, functions take objects as arguments and manipulate them to produce an object, which is returned. There are two caveats to this general principle.

1. A function may have side effects, such as printing some output to the console or producing a plot. Indeed a function may only produce a side effect, and no return object. Generally side effects that modify objects that are external to the function are to be avoided, if code is to be clean and easy to debug.

2. A function may make use of objects not in its argument list: if R encounters a symbol not in the function argument list and not previously created within the function, then it searches for it, first in the environment in which the function was *defined*[7] (which is not necessarily the environment from which it was called). If that fails it looks in the environments returned by function `search()`. A benign use of this mechanism is to call other functions not in a function's argument list, or to access constants such as those stored in `.Machine`. Using this mechanism to provide a function with other objects that you have created is generally bad practice, because it makes for complex hard-to-debug code. Generally all objects that a function needs should be provided as its arguments. If this gets unwieldy, then group the arguments into a smaller number of list arguments.

**特征分解(当A是对称矩阵时)**

Here is an example of a function definition. It generalises one-to-one real functions with power series representations to symmetric matrices using the following idea. The eigen-decomposition of symmetric matrix $\mathbf{A}$ is $\mathbf{A} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^{\mathrm{T}}$ where the columns of $\mathbf{U}$ are eigenvectors of $\mathbf{A}$ and $\mathbf{\Lambda}$ is the diagonal matrix of eigenvalues. The generalization of a function, $f$ is then $f(\mathbf{A}) = \mathbf{U}f(\mathbf{\Lambda})\mathbf{U}^{\mathrm{T}}$, where $f$ is applied element wise to $\mathbf{\Lambda}$'s diagonal.

```
mat.fun <- function(A,fun=I) {
  ea <- eigen(A,symmetric=TRUE)
  ea$vectors %*% (fun(ea$values)*t(ea$vectors)) ## note use of re-cycling rule!
}
```

recycling-rule

'`function(A,fun=I)`' indicates that a function is to be created with arguments `A` and `fun`. In this case the function created is given the name `mat.fun`, but functions are sometimes used without being given a name (for example, in the arguments to other functions). The argument list gives the names by which the function arguments

[6] These inequalities would be reversed for a decreasing function, of course

[7] This is known as 'lexical scoping', because the parent environment of the function is where it was written down.

will be referred to within the function body. Arguments may be given default values to be used in the event that the function is called without providing a value for that argument. This is done using `name = default` in the argument list. `fun=I` is an example of this, setting the default value of `fun` to the identity function.

Next comes the body of the function given by the R expressions within the curly brackets `{ ... }` (if the function body consists of a single expression, then the brackets are not needed). The function body can contain any valid R expressions. The object created on the last line of the function body is the object returned by the function. Alternatively the object can be returned explicitly using the `return` function. For `mat.fun`, the eigen decomposition of the first argument is obtained and then used to produce the generalised version of `fun`.

Now let us use the function, with a random matrix. First a sanity check that the identity function is correct:

```
> set.seed(1)
> m <- 3; B <- crossprod(matrix(runif(m*m),m,m)) ## example matrix
> range(B - mat.fun(B)) ## check input matches output
[1] -2.220446e-16 6.661338e-16
```

This confirms that the output matches the first argument when the default identity function is used.

An aside: why was the difference between the input and output not exactly 0? Because real numbers can only be stored in a computer to a finite precision, here equivalent to about 16 places of decimals. This inevitably means that arithmetic computations on real numbers are not exact — rounding errors accumulate as calculations are performed. A great deal of mathematical work has gone into minimising these numerical errors in matrix computations, but they can not be eliminated. In this case we can say that the input and output matrices are identical *to machine precision*. You can get an idea of the size of number that counts as indistinguishable from zero by typing `.Machine$double.eps`. Note, however that what counts as machine zero is relative to the size of numbers involved in a calculation. For example, here is what happens if `B` is multiplied by $10^{10}$

```
> B <- B * 1e10
> range(B - mat.fun(B))
[1] -3.814697e-06 3.814697e-06
```

Back to functions! What actually happened when the function was called (by `mat.fun(B)`). R first matches the arguments of the function to those actually supplied, adopting a rather permissive approach to so doing. First it matches on the basis of exact matches to argument names ('`A`' and '`fun`' in the example). This does not mean that R is looking for `B` to be called `A` in the example; rather it is looking for statements of the form `A=B`, specifying unambiguously that object `B` is to be taken as argument '`A`' of `mat.fun`. After exact matching, R next tries partial matching of names on the remaining arguments; for example `mat.fun(B,fu=sqrt)` would cause the `sqrt` function to be taken as the object to be used as argument `fun`. After matching by name, the remaining arguments are matched by position in the argument list: this is how R has actually matched `B` to `A` earlier. Any unmatched argument is matched to its default value.

R next creates an *evaluation frame*: an extendible piece of memory in which to store copies of the function arguments used in the function, as well as the other objects created in the function. This evaluation frame has the environment of the function as its parent (which is the environment where the function was defined, remember).

Having matched the arguments, R does not actually evaluate them immediately, but waits until they are needed to evaluate something in the function body: this is known as *lazy evaluation*. Evaluation of arguments takes place in the environment from which the function was called, except for arguments matched to their default values, which are evaluated in the function's own evaluation frame.

Preliminaries over, R then evaluates the commands in the function body, and returns a result.

Notice that arguments are effectively copied into the function's evaluation frame, so nothing that is done to a function argument within the function has any effect on the object that supplied that argument 'outside' the function. Within the body of `mat.mod` argument `A` could have been replaced by some poetry, but the matrix `B` would have remained unaltered.

Here is an example of calling `mat.mod` to find a matrix inverse:

```
> mat.fun(A = B, fun = function(x) 1/x)
         [,1]     [,2]     [,3]
[1,] 10.108591 -2.164337 -3.070143
[2,] -2.164337 4.192241 -2.707381
[3,] -3.070143 -2.707381 4.548381
```

匿名函数

将 fun 设为 1/x，mat.fun 可以用于求矩阵的逆

In this case both arguments were supplied by their full name, and a function definition was used to supply argument `fun`.

## 5.7 The '. . .' argument

Functions can also have a special argument '. . .', which is used to create functions that can have variable numbers of arguments. It is also used to pass arguments to a function that may in turn be passed on to other functions, without those arguments having to be declared as arguments of the calling function: this is useful for passing arguments that control settings of plotting functions, for example.

Any arguments supplied in the call to a function, that are not in the argument list in the function definition, are matched to its '. . .' argument, if it has one.[8] The elements of '. . .' can be extracted into a list, to work with, but here we will just consider how to use '. . .' to pass arguments to a function called inside another function *without having to know the names of the arguments in advance*.

To make the problem concrete, suppose that we want to use our `mat.fun` function with the function argument defined by

```
foo <- function(x,a,b) x/a + b
```

Obviously `foo` can not be used directly with `mat.fun` as the code assumes that whatever is passed in as `fun` only has one argument. We certainly don't want to write a new version of `mat.fun` just for 3 argument functions with arguments called `a` and `b`. That's where '. . .' comes to the rescue: we'll use it to pass `a` and `b`, or any other extra arguments (or none) to `fun`, as follows

```
mat.fun <- function(A,fun=I,...) { ## ... allows passing of extra arguments
  ea <- eigen(A,symmetric=TRUE)
  ea$vectors %*% (fun(ea$values,...)*t(ea$vectors)) ## to be passed to fun
}
```

And it works...

```
mat.fun(B,fun=foo,a=2,b=3)
         [,1]     [,2]     [,3]
[1,] 2685660117 4154166277 4285541075
[2,] 4154166277 8363105089 7782109902
[3,] 4285541075 7782109902 8624247833
```

One irritation is worth being aware of.

```
ff <- function(res=1,...) res;f(r=2)
```

will return the answer 2 as a result of partial matching of argument names, even if you meant `r` to be part of the '. . .' argument. It is easy to be caught out by this. If you want '. . .' to be matched first, then it has to precede the arguments it might be confused with. So the following gives the answer 1:

```
ff <- function(...,res=1) res;f(r=2)
```

```
root_find <- function(f, interval, tol = 1e-8, ...) {
   x0 <- interval[1]
   x1 <- interval[2]
   if (f(x0, ...) * f(x1, ...) > 0) stop("The root is
not bracketed.")

   repeat {
      xt <- (x0 + x1) / 2
      if (abs(f(xt, ...)) < tol) return(xt)
      if (f(xt, ...) * f(x0, ...) < 0) {
         x1 <- xt
      } else {
         x0 <- xt
      }
      if (abs(x1 - x0) < tol) return(xt)
   }
}
```

### Exercise

Turn your code from the section 5.5 exercise 2 into a general root finding function that takes an initial interval, a function (which may depend on some extra constants) the root of which is to be found (that is the value of its argument at which the function is zero). Try your function out with the function from the previous exercise, and with $g(x) = x^b - e^{ax} + 2$ where $b \geq 1$ and $a > 0$. Make sure that your function checks that the initial supplied interval brackets the solution, and can deal with increasing or decreasing functions. Function `stop` can be used to jump out of a function and print an error message if a problem is detected. If your function does not work as expected, try stepping through it using the `mtrace` from the `debug` package to figure out where the error is.

[8] This has the slightly unfortunate side effect that mistyped argument names do not generate obvious warnings.

## 5.8 Pipes

Suppose that we want to generate 1000 random deviates from a $N(0, 0.5^2)$ distribution, exponentiate them, and plot a histogram of the result. A single line of R code would accomplish this task using nested function calls.

```
hist(exp(rnorm(1000,sd=.5)))
```

However the more levels of nesting there are the more awkward this can be to read. An obvious alternative is

```
z <- rnorm(1000,sd=.5)
x <- exp(z)
hist(x)
```

Code written like this is much easier to read *and much easier to debug if something goes wrong*. But it is also devoting several lines of code to a rather trivial task. Writing in this way can make the code for a more complex task become rather lengthy. That in turn can make the overall structure of the code more difficult to follow.

For this reason it is sometimes useful to use *pipes*, to feed the results from one function directly to the arguments of another. In R the pipe operator is `|>` (see `?"|>"` or `?pipeOp`). Here it is used for the same simple example

```
rnorm(1000,sd=.5) |> exp() |> hist()
```

The result of the function call `rnorm(1000,sd=.5)` is a vector of 1000 numbers. The pipe operator takes that vector and feeds it in as the first (in this case only) argument of function `exp`. Function `exp()` produces a vector of 1000 transformed numbers and the second pipe operator feeds that in as the first argument of `hist()`.

The functions in a piped sequence can be supplied with other arguments, and you can also pipe to an argument other than the first using the _ placeholder. The following illustrates these features. It generates quantiles of a lognormal density and plots the ordered simulated transformed data against these.

```
n <- 1000    |> 的优先级高于除法
((1:n-.5)/n) |> qlnorm(,sdlog=.5) -> q
rnorm(n,sd=.5) |> exp() |> sort() |> plot(q,y=_,ylab="obs")
```

Notice how additional named arguments have been supplied to `qlnorm` and `plot`. Notice also how `y=_` has been used to indicate that the result of `sort()` is to be piped to the second argument of `plot`. Finally note the enclosing brackets in `((1:n-.5)/n)`. Why are these necessary? Because `|>` has higher precedence than `/`. Without the brackets `n` is fed in as the first argument of `qlnorm` generating an `NA`, by which the vector `1:n-.5` is then divided – not what is wanted at all.

Piping has become very popular, as it often nicely expresses how we think about sequential data processing tasks. However with popularity has come overuse, for tasks where it obscures more than it illuminates. Code consisting of half a page or more of sequential piping is rarely readable, and is all but impossible to debug if something goes wrong. For easy debugging we need code broken down into small steps, preferably with intermediate quantities readily accessible: like explicit nested function calls piping does not provide this. Use it thoughtfully.

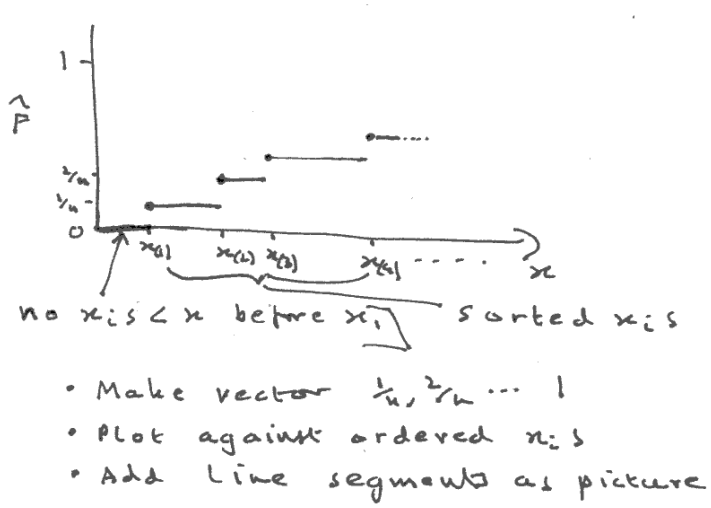
## 5.9 Planning and coding example: plotting an empirical CDF

The cumulative distribution function, $F$, of a random variable $X$ is defined as $F(x) = \Pr(X \leq x)$. Hence if we have a random sample $x_1, x_2, \ldots, x_n$ of observations of $X$ we can define an *empirical* cumulative distribution function

$$\hat{F}(x) = \frac{1}{n} \sum_{i=1}^{n} \mathbb{I}(x_i \leq x)$$

要么为1，要么为0

where $\mathbb{I}$ is the indicator function. Suppose we want to write a function to take a sample of data in a vector $\mathbf{x}$ and plot its CDF. There are many ways to do this, so the first thing we have to do is to write down a plan. Usually this will involve jotting down ideas and maybe sketches on a piece of paper. Here are my notes for this task...



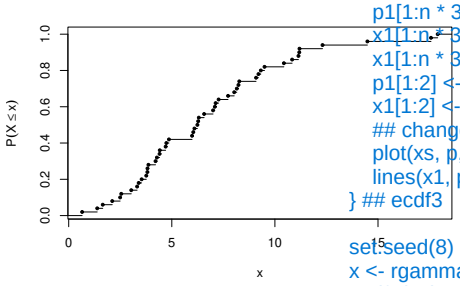

基于每个样本点 x_i 发生的概率相同，
该图表示了抽取到 x_i **小于等于** x **的概率**

The sketch of the CDF is me trying to get my head around what the definition of $\hat{F}$ really means, and whether the $x_i$ values are at the start of a step or just before it (the former). As a result of this figuring out, I get to a plan that is a bit better than just evaluating the formula for $\hat{F}$ for a finely spaced sequence of $x$ values. Obviously more complicated tasks may involve several sheets of paper, false starts, and the need to summarize the design at the end. Code that involves no sheets of paper and none of this planning process is often poorly designed, error prone and slow to write.

To implement the design I used built in functions, `sort`, `plot` and `lines`. Look up the help pages for these. `plot` is a high level function for producing scatter plots, while `lines` is a lower level function for adding lines to an existing plot (`points` is an equivalent for adding points). For a little extra flourish I used `expression` for defining the y axis label — see `?plotmath` for how that works. Here is the code – makes sure you understand each line.

```
ecdf <- function(x) {
## function to plot empirical cdf of sample x
  n <- length(x)
  p <- 1:n/n ## create cumulative probability vector
  xs <- sort(x) ## sort the x values
  ## plot the x, p points
  plot(xs,p,pch=19,cex=.5,xlab="x",ylab=expression(P(X<=x))) expression() 来生成数学符号
  ## now add in the lines of constant probability between x values
  lines(c(2*xs[1]-xs[n],xs[1]),c(0,0))
  for (i in 1:(n-1)) lines(c(xs[i],xs[i+1]),c(p[i],p[i]))
  lines(c(2*xs[n]-xs[1],xs[n]),c(1,1))
} ## ecdf

## test it
set.seed(8);
x <- rgamma(50,shape=3,scale=2) ## test data
ecdf(x)
```




```
ecdf3 <- function(x, h = .1) {
  ## function to plot empirical cdf of sample x, smoothed with
  ## bandwidth h
  x <- x + rnorm(length(x) * 100, sd = h) ## changed line!!
  n <- length(x)
  p <- 1:n / n ## create cumulative probability vector
  xs <- sort(x) ## sort the x values
  ## plot the x, p points
  p1 <- x1 <- rep(NA, 3 * n + 2) ## create p1 and x1 with NAs everywhere
  p1[1:n * 3 + 2] <- p1[1:n * 3 + 1] <- p ## fill in the step heights
  x1[1:n * 3 + 1] <- xs ## step start
  x1[1:n * 3 + 2] <- c(xs[-1], 2 * xs[n] - xs[1]) ## step ends
  p1[1:2] <- 0 ## initial prob is zero
  x1[1:2] <- c(2 * xs[1] - xs[n], xs[1]) ## x co-ords for zero prob
  ## changed plot symbol!!...
  plot(xs, p, pch = ".", cex = .5, xlab = "x", ylab = expression(P(X <= x)), col = 4)
  lines(x1, p1, col = 2)
} ## ecdf3

set.seed(8)
x <- rgamma(50, shape = 3, scale = 2) ## test data
ecdf3(x, h = 1.5)
x0 <- seq(0, 25, length = 200)
lines(x0, pgamma(x0, shape = 3, scale = 2))
```

### 5.9.1 Avoiding the loop and vectorization

You will notice that `ecdf` uses a `for` loop for plotting each line segment in turn, but actually if you check the help page `?lines` you will see that `lines` has a mechanism for plotting multiple separate line segments all at once - we just separate the co-ordinates defining the lines with an `NA` (not available) to indicate that the points either side of the `NA` should not be joined by a line. Since vectorized code is usually faster than loop code, lets think about how to uses this feature.

1. For $n$ data we have $n + 1$ line segments, each defined by 2 $x, y$ points, and between each line segment we will need `NA` values ($n$ in total).

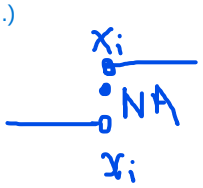
2. So let's create vectors, `p1` and `x1` each of length $3n + 2$ to contain the line segment definitions and breaks, and fill them in.

Here is the code that could replace everything after *`## plot the x, p points`* in `ecdf`.

```
p1 <- x1 <- rep(NA,3*n+2) ## create p1 and x1 with NAs everywhere
p1[1:n*3+2] <- p1[1:n*3+1] <- p ## fill in the step heights
x1[1:n*3+1] <- xs ## step start
x1[1:n*3+2] <- c(xs[-1],2*xs[n]-xs[1]) ## step ends
p1[1:2] <- 0 ## initial prob is zero
x1[1:2] <- c(2*xs[1]-xs[n],xs[1]) ## x co-ords for zero prob
plot(xs,p,pch=19,cex=.5,xlab="x",ylab=expression(P(X<=x)))
lines(x1,p1)
```

p1 = (0, 0, NA, p[1], p[1], NA, .....)

x1 = (2*xs[1] - xs[n], x[1], NA, x[1], x[2], NA, .....)




Notice how this code is entirely vectorized — no repeating of code $n$ times in a loop. It is therefore faster to run. But also notice that it is slower to read and understand. Also, with 500 data points the vectorized version of `ecdf` is about 50 times faster than the original version. But it still takes only 0.7 seconds to run on my machine. So was it worth the effort of speeding up? That really depends on how often the function will be used and on what size data set. These are common trade-offs, which mean that you should beware of making a fetish out of vectorizing your code. Always vectorize if it is easy to write and understand the vectorized code, but otherwise think about whether the computer time savings will justify the effort.

## Exercise

One approach to getting a smoother approximation to the CDF is to recognize that we are effectively using the sorted data to approximate the quantiles of the distribution, and that there is uncertainty in these quantile estimates - the quantiles could just as well have been a bit higher or lower. This suggests that we could generate many replicate datasets in which we add noise to the original data, combine these simulated datasets and plot their CDF, to get a smoothed CDF. Modify the (vectorized) `ecdf` function to do this, by generating 100 replicated datasets, each obtained by adding normal noise (`rnorm`) to the original data, with standard deviation given by $h$ (default 1), an argument of the modified function. You probably want to change the default plot character to `"."`.

## 5.10 Useful built-in functions

R has a huge number of useful built in functions. This section is about how to find them. Recall that R's extensive help system can be accessed by typing `help.start()` at the command prompt, to obtain help in navigable HTML

form, or by typing `?foo` at the command line, where `foo` is the function or other topic of interest.

| Help topic | Subject covered |
|---|---|
| `?Arithmetic` | Standard arithmetic operators |
| `?Logic` | Standard logical operators |
| `?sqrt` | Square root and absolute value functions |
| `?Trig` | Trigonometric functions (`sin`, `cos`, etc.) |
| `?Hyperbolic` | Hyperbolic functions (`tanh`, etc.) |
| `?Special` | Special mathematical functions ($\Gamma$ function, etc.) |
| `?pgamma` | Partial gamma function |
| `?Bessel` | Bessel functions |
| `?log` | Logarithmic functions |
| `?max` | Maximum, minimum and vectorised versions |
| `?round` | Rounding, truncating, etc. |
| `?distributions` | Statistical distributions built into R |

The `?distributions` topic requires some more explanation. R has built-in functions for the `beta`, `binomial`, `cauchy`, `chisq`uared, `exp`onential, `f`, `gamma`, `geom`etric, `hyper`geometric, `lnorm`al (log-normal), `multinom`ial, `nbinomial` (negative binomial), `norm`al, `pois`son, `t`, `unif`orm and `weibull` distributions. The R identifying names for these are shown in `courier` font in this list.

For each distribution, with name `dist`, say, there are four functions:

1. `ddist` is the probability (density) function of `dist`.
2. `pdist` is the cumulative distribution functions of `dist`.
3. `qdist` is the quantile function of `dist`.
4. `rdist` generates independent pseudorandom deviates from `dist`.

## 5.11 The `apply` functions

Often the same operation has to be applied over one or more margins of an array (e.g. over the rows or the columns of a matrix), or over the elements of a list, or over subsets of some vector. R functions `apply`, `lapply` and `tapply` achieve this. Here are some examples.

```
a <- array(1:24,c(2,3,4)) ## creating an example 2 by 3 by 4 array
```

Let's apply the `sum` function to the set of elements sharing each value (i.e. 1 and 2) of the first index of `a`

```
apply(a,1,sum)
# [1] 144 156
```

Or perhaps we would like to apply the `prod` function to all elements of `a` sharing each combination of its first and third indices

```
apply(a,c(1,3),prod)
#     [,1] [,2] [,3] [,4]
# [1,]   15  693 3315 9177
# [2,]   48  960 4032 10560
```

For vectors (and remember that the multi-dimensional arrays can always be accessed as vectors as well) it is often the case that functions should be applied to irregular subsets of the vector. This is easily achieved by assigning the level of a factor variable to each subset and using `tapply`. For example let's sum the integers 1 to 10, divided into 3 groups.

```
x <- 1:10          1, 2, 3, 4, 5, 6, 7, 8, 9, 10
fac <- factor(c(1,2,2,3,1,2,2,3,1,1))   一一对应，进行分组
tapply(x,fac,sum)
# 1  2  3
# 25 18 12
```

It is not unusual to require groups defined by combinations of factors. Here a second factor is introduced, and `sum` applied to the set of elements corresponding each unique combination of factor levels.

```
fac2 <- factor(rep(c("a","b"),each=5))
tapply(x,list(fac2,fac),sum)
#    1   2   3
# a  6   5   4
# b 19  13   8
```

x 中既是标签"3"又是标签"a"的元素的和

apply(): 用于矩阵或数组，按行或按列应用函数。
lapply(): **用于列表或向量，**对每个元素应用函数，**返回列表。**
sapply(): **用于列表或向量，**对每个元素应用函数，**返回**简化后的结果（向量或矩阵）。
tapply(): **用于向量，**按因子分组对子集应用函数。
mapply(): **用于多个列表或向量，**对每个位置的元素应用**函数。**

An `NA` results for any factor combinations not occurring in the data

`lapply` is even simpler. The same function is applied to each element of a list. Here is an example illustrating its application to a list containing vectors of different lengths, where the second smallest entry of each is required. The example also illustrates the supply of a custom function (to any of the `apply` functions).

```
b <- list(a=1:5,b=c(4,2,7),c=-10:10)
unlist(lapply(b,function(x) sort(x)[2]))
# a b  c
# 2 4 -9
```

`lapply` returns a list of the same length as the input list. Here `unlist` has been used to turn this into a vector. `sapply` is an alternative to `lapply` that returns a vector by defualt.

### Exercise

Run the following code to create a list of matrices

```
set.seed(4);M<-list();for (i in 1:10) M[[i]] <- matrix(runif((i+1)^2)-.5,i+1,i+1)
```

Complete the following, without using any loops.

1. Give the elements of `M` names `M2`, `M3` ...`M11`. Use `paste`.
2. Obtain a vector of the Frobenius `norm` for each element of `M`.
3. Obtain a list giving the singular value decomposition (`?svd`) of each element of `M`.
4. Using `apply`, write a function to form $\alpha = \prod_i \max_j |A_{ij}|$ for a matrix $\mathbf{A}$ (i.e. the product of the maximum absolute value in each column).
5. Obtain the vector of $\alpha$ values for each element of `M`

## 6 Simulation I: stochastic models and simulation studies

One major class of statistical programming task involves *stochastic simulation*. Simulating processes that involve random numbers. Strictly speaking *pseudorandom* numbers. There are several main sorts of simulation that we might need to undertake.

1. Simulate from a stochastic model. e.g. a model of disease spread, or a model of football teams competing, or a model of how shoppers will move around a store.
2. Simulate the process of sampling data from a population, and analysing it using a statistical method, in order to assess how well the statistical method is working.
3. Simulate the process of sampling data directly, as part of a statistical analysis method.
4. Simulate from the posterior distribution of the parameters in a Bayesian data analysis.

This section will take a look at examples of the first two.

## 6.1 Random sampling building blocks

There are a couple of sorts of random sampling that we might want to do:

1. Sample observations from a set of data.
2. Sample random deviates from some particular probability distribution.

### 6.1.1 Sampling data

Suppose `x` is a vector and we would like to sample `n` of its elements, at random, without replacement. `sample` is a function for doing this. For example, let's draw a random sample of size 10 from the integers $1, 2, \ldots, 100$.

```
> set.seed(0) ## just so you can reproduce what I got
> sample(1:100,10) ## sample(100,10) is equivalent - be warned!! See ?sample.
 [1] 14 68 39  1 34 87 43 100 82 59
```

This sampled each number with equal probability, *without replacement*. So each observation had the same chance of occurring in the sample, but could do so only once. This is the sort of sampling that you get if you put numbered balls in a bag, shake them up and then draw out a sample from the bag without looking.

Notice the use of `set.seed()`. We can not generate truly random numbers, but only sequences of numbers that appear indistinguishable from random using any statistical test you might apply. What ever sort of random number we eventually want, R starts by generating sequences that appear to be uniform random deviates from $U(0, 1)$, and then transforming these. These uniform sequences are generated using a *random number* generator, and are in fact completely deterministic. If we repeatedly start the sequence off from the same state, it will generate the same sequence every time. Setting the random number generator to a particular state is known as *setting the seed*. To fix ideas, here is how it works for the basic uniform generator

```
> set.seed(3);runif(5) ## 5 random numbers after setting seed
[1] 0.1680415 0.8075164 0.3849424 0.3277343 0.6021007
> runif(5) ## another 5 - all different
[1] 0.6043941 0.1246334 0.2946009 0.5776099 0.6309793
> set.seed(3);runif(5) ## and after setting the seed to 3 again - same as first time
[1] 0.1680415 0.8075164 0.3849424 0.3277343 0.6021007
```

Let's get back to sampling. We might also want to sample *with replacement*. This is as if we sampled numbered balls from a bag, as before, but this time after noting the number we return the ball to the bag and give the bag a shake, before drawing the next ball.

```
> sample(1:100,10,replace=TRUE)
 [1] 51 97 85 21 54 74 7 73 79 85
```

Notice that 85 occurred twice in this sample. Obviously we can sample any vector in this way, not just a vector of integers, but sampling integers like this is especially convenient if you want to sample whole rows of data from a matrix or data.frame. You sample the row indices, and then use these to extract the sampled rows. For example, suppose we want a random sample of 30 rows of data frame, `dat`:

抽取数据框的行

```
ii <- sample(1:nrow(dat),30) ## 'nrow' returns the number of rows of 'dat'
dat1 <- dat[ii,]      ## 30 rows selected at random from 'dat'
```

It is also possible to sample with different probabilities of sampling each observation. For example let's sample the integers $i = 1, 2, \ldots, 100$ each with probability $\propto 1/i$.

```
> sample(1:100,10,prob=1/1:100)
 [1] 13 1 5 19 24 14 12 17 20 63
```

用一一对应的向量设置概率

Note that we are sampling with probability proportional to `prob` – `sample` will normalise the probabilities so that they sum to one internally, before using them. In this example we only got one observation $> 25$. Does that seem as expected? Let's work out the probability of sampling an observation $< 25$ as a sanity check...

```
> sum(1/1:24)/sum(1/1:100)
[1] 0.7279127 ## seems fine!
```

抽取到1:24的概率

**Example: the birthday problem**

A standard toy problem in probability is the birthday problem. What is the probability of at least 2 people in a group of 30 sharing the same birthday? Let's investigate this by simulation. As usual we start by planning. For each group we want to sample 30 birthdays at random from the numbers $1 \ldots 366$, with one of the numbers occurring only in leap years and therefore having 1/4 of the probability of the others. For each group we can then find the unique birthdays, and see how often there are fewer than 30 of them. So

1. Randomly sample the $n$ lots of 30 birthdays, and put them in an $n \times 30$ matrix.
2. For each matrix row, count the unique birthdays.
3. Find the proportion for which that count is $< 30$

Here is an implementation, it uses the `apply` functions to apply a function counting the unique birthdays to each row of the birthday matrix. The second argument of `apply` gives the dimensions of the first argument over which the function should be applied. Here the first argument is a matrix, so I give one as the second argument, to apply over rows.

```
n <- 1000000 ## number of groups
bd <- matrix(sample(1:366,30*n,replace=TRUE,prob=c(rep(4,365),1)),n,30) ## birthdays
p <- mean(apply(bd,1,function(x) length(unique(x)))!=30)
```

要么为 True（1），要么为 False（0）

It gives the answer p=0.706. Obviously there is some stochastic variability in that answer - we could work out the standard error of the answer using the variance of a Bernoulli random variable: `(p*(1-p)/n)^.5` is about 0.0005. Or if we are lazy we could just run the simulation repeatedly to assess the variability of the answer.

标准误差公式（样本均值的标准差）

每一个三十人的实验可以看成是一个伯努利随机变量

There are two rather appealing features of this way of answering the question. Firstly, it was very easy, in a way that most people do not find the probability calculation based answer. More importantly, it is very easy to generalize to situations for which the exact answer would be very difficult to work out. For example, births are not spread out evenly throughout the year, so the probabilities of the different birthdays are not all equal. Given appropriate data it would be easy to adjust the probabilities in the sampling step, and get an adjusted probability for that case too.

**Exercise**

*Bootstrapping* is the process of resampling with replacement from a set of data, in order to simulate the process of sampling the data from the population, in order to assess the sampling variability in quantities calculated from the data (statistics). In R `morley$Speed` contains the measured speed of light in km/s above 299000 km/s, in 100 runs of the Michaelson Morley experiment. Sample these data with replacement to generate 1000 bootstrap replicate datasets. Find the mean of each replicate, and then use the `quantile` function to find the range of the middle 95% of the bootstrap means. This range is sometimes known as a *bootstrap percentile confidence interval*. You may find `colMeans` or `rowMeans` useful.

1000组数据

```
n <- length(morley$Speed)
bs.ls <- matrix(sample(morley$Speed, n * 1000, replace = TRUE), n, 1000)
bs.mean <- colMeans(bs.ls)
quantile(bs.mean, c(.025, .975))
```

### 6.1.2 Sampling from distributions

R has functions for simulating random variables from a wide variety of basic distributions: see `?distributions` for an overview. The functions all start with an `r` followed by the distribution name. Their first argument is the number of values to simulate, and remaining arguments are vectors of distribution parameters. For example, if I would like 3 normal random deviates, with means 1, 3.2 and -0.5 and standard deviations 1, 1 and 2.5 then the following produces them

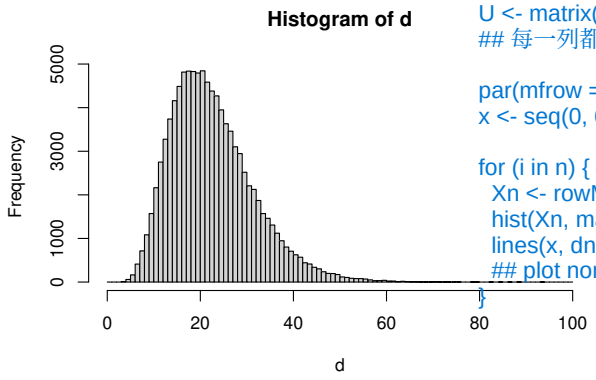
```
> rnorm(3,mean=c(1,3.2,-.5),sd=c(1,1,2.5))
[1] 0.4268462 2.1288700 2.3638236 ## you should get different answers here!
```

An immediate use for simulation is to quickly get approximate probabilities that may be awkward to compute otherwise.

For example the time from infection to onset of Covid-19 symptoms follows a log normal distribution with log scale mean and sd of 1.63 and 0.50. Meanwhile one estimate has the distribution of time from symptom

onset to death estimated to be a gamma distribution with shape parameter 4 and scale parameter 4.2. Assuming independence of the 2 durations, what does the distribution of time from infection to death look like, and what is the probability that it is less than one week?

```
> n <- 100000
> d <- rlnorm(n,1.63,.5) + rgamma(n,shape=4.45,scale=4)
> mean(d<7)
[1] 0.00389
> hist(d,breaks=0:100) ## plots a histogram with bar breaks as given
```




```
n <- 2^(1:9) ## the n sequence
m <- 10000 ## sample sizes

U <- matrix(rexp(m * max(n)), m)
## 每一列都是一个指数分布的随机变量

par(mfrow = c(3, 3)) ## split the plot window
x <- seq(0, 6, length = 1000)

for (i in n) { ## loop over n
  Xn <- rowMeans(U[, 1:i]) ## 前 i 个随机变量的均值
  hist(Xn, main = "", xlab = paste("X", i), freq = FALSE, breaks = 30)
  lines(x, dnorm(x, mean = mean(Xn), sd = sd(Xn)))
  ## plot normal pdf with same mean and sd
}
```

**Exercise**

The *Central Limit Theorem* states that the distribution of the sum or mean of a set of $n$ independent identically distributed finite variance random variables tends to a normal distribution as $n \to \infty$. Produce an illustration of this by simulating random variables $X_n = n^{-1} \sum_i^n U_i$ where the $U_i$ are i.i.d. exponential random variables, with rate parameter 1 (see `?rexp`). For $n = 2^1, , 2^2, \ldots, 2^9$ produce histograms illustrating the distribution of 10000 simulated values of each $X_n$. `par(mfrow=c(3,3))` will set up a graphics window to show all the histograms on the same page.

## 6.2 Simulation from a simple epidemic model

Let's put a few things together and build a slightly more complicated simulation. SEIR (Susceptible-Exposed-Infectious-Recovered[9]) models are commonly used in disease modelling. In the simplest version almost everyone starts out susceptible, and then move to the exposed class according to their daily probability of being infected by someone in the infectious class. This daily probability is assumed proportional to the number of people in the infectious class, with constant of proportionality $\beta$. People move from the exposed to infectious class with constant probability, $\gamma$ , each day, and move from the infectious to the R class with a different constant probability, $\delta$.

Suppose that we want to simulate from this stochastic SEIR model, maintaining a record of the populations in the different classes over time. And suppose in particular that we are interested in investigating scenarios in which people's daily probability of catching the disease is variable from person to person, so that each has a different $\beta$ value. In fact let's assume that each person has their own $\beta_i$ drawn from a gamma distribution. As usual we first need to think about code design.

1. Since the code is for investigating multiple scenarios, we should produce a function that takes parameter values and control constants as inputs, and outputs vectors of the S, E, I and R populations over time.

2. We could maintain vectors for the different stages, and move individuals between them, but this seems clumsy. It is probably better to have a single vector maintaining an integer state indicating each individual's current status 0 for S, 1 for E, 2 for I and 3 for R.

[9] often a euphemism for recovered or dead

3. The individual states are then updated daily according to the model, and the numbers in each state summed.

Here is an implementation.

```
seir <- function(n=10000,ni=10,nt=100,gamma=1/3,delta=1/5,bmu=5e-5,bsc=1e-5) {
## SEIR stochastic simulation model.
## n = population size; ni = initially infective; nt = number of days
## gamma = daily prob E -> I; delta = daily prob I -> R;
## bmu = mean beta; bsc = var(beta) = bmu * bsc
  x <- rep(0,n) ## initialize to susceptible state
  beta <- rgamma(n,shape=bmu/bsc,scale=bsc) ## individual infection rates
  x[1:ni] <- 2 ## create some infectives
  S <- E <- I <- R <- rep(0,nt) ## set up storage for pop in each state
  S[1] <- n-ni;I[1] <- ni ## initialize
  for (i in 2:nt) { ## loop over days
    u <- runif(n) ## uniform random deviates
    x[x==2&u<delta] <- 3 ## I -> R with prob delta
    x[x==1&u<gamma] <- 2 ## E -> I with prob gamma
    x[x==0&u<beta*I[i-1]] <- 1 ## S -> E with prob beta*I[i-1]
    S[i] <- sum(x==0); E[i] <- sum(x==1)
    I[i] <- sum(x==2); R[i] <- sum(x==3)
  }
  list(S=S,E=E,I=I,R=R,beta=beta)
} ## seir
```

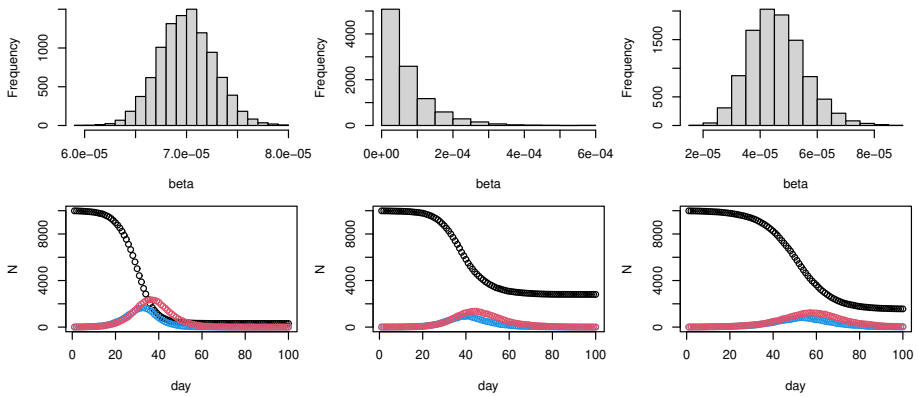
Hopefully by now most of this should be relatively easy to understand. The lines like

```
x[x==2&u<delta] <- 3
```

are doing most of the real work, so these are especially important to understand. First note that elements of the logical vector `u<delta` will be `TRUE` with probability `delta` and `FALSE` otherwise. This is because the elements of `u` are $U(0,1)$ deviates and hence have probability `delta` of being less than `delta`. Elements of `x==2&u<delta` are then `TRUE` only when the corresponding elements of `u<delta` **and** `x==2` are `TRUE`. As a result `x[x==2&u<delta] <-3` takes each element of `x` in state 2 and with probability `delta` sets it to state 3. The other lines act similarly, with the probability for the state 0 to 1 transition varying between elements of `x`.

The fixed $\beta$ version of this basic epidemic model is close to 100 years old. A great deal of the modelling work on Covid uses variations on this structure, perhaps surprisingly, given the simplistic way that the infection process is modelled. Even more surprisingly most of the models ignore the fact that there is wide variability in the number of contacts people have with other people each day, and in their chance of becoming infected at each contact – i.e $\beta$ is not treated as random. We can use our simple implementation to investigate whether this matters. In particular consider a simulation with almost no variability in $\beta$, with `bmu` and `bsc` set to `7e-5` and `1e-7`. Then compare that to a simulation in which $\beta$ varies more widely (`bsc=7e-5`). Finally consider a universal social distancing simulation, in which the mean and variability in $\beta$ are reduced (`bmu=5e-5`, `bsc=2e-6`).

```
par(mfcol=c(2,3),mar=c(4,4,1,1)) ## set plot window up for multiple plots
epi <- seir(bmu=7e-5,bsc=1e-7) ## run simulation
hist(epi$beta,xlab="beta",main="") ## beta distribution
plot(epi$S,ylim=c(0,max(epi$S)),xlab="day",ylab="N") ## S black
points(epi$E,col=4);points(epi$I,col=2) ## E (blue) and I (red)
...
```




Comparing the left two columns we see that neglecting the variability in $\beta$ means that the severity of the epidemic will be overestimated by the model. Comparing the right two columns we see that social distancing slows the epidemic down, but can make it larger in the long run, by suppressing the variability in $\beta$ (as everyone has to follow the same rules). Perhaps most surprising of all, scientists who tried to point out the problems with neglecting variability in $\beta$ had extreme difficulty getting published, and were often vilified on social media. Some models used for policy did represent the differences in contact rates between different age groups. But the main survey used to do this was based on one day diaries kept by survey participants of whom 7 were over 75 and none over 80. This is quite limited data for the age groups that make up 3/4 of Covid deaths to date.

## Exercise

Try stepping through the `seir` code line-by-line using package `debug` (`library(debug)` loads it). `mtrace(seir)` indicates that you want to step into this function. Then call `seir` as usual, and you will execute it in the debugger. Pressing the return key steps forward. `bp` is used to set breakpoints. `go()` runs until completion, or the next breakpoint. `https://www.maths.ed.ac.uk/~swood34/RCdebug/RCdebug.html` has more information on debugging. `mtrace(seir, FALSE)` to stop debugging `seir`. Or `mtrace.off()` to stop all debugging.

## 6.3 Simple simulation study: testing an approximate confidence interval

Suppose we observe a binomial random variable $X \sim \text{binom}(n, p)$ (number of successes in $n$ independent trials, each with probability of success $p$). $\hat{p} = X/n$ is an obvious estimator, of $p$, and a simple plug in estimate of its variance is then $\hat{\sigma}_p^2 = \hat{p}(1-\hat{p})/n$, suggesting approximate 95% confidence interval $\hat{p} \pm 1.96\hat{\sigma}_p$. The central limit theorem implies that this interval will have the correct coverage probability in the $n \to \infty$ limit, but how will it behave when $n = 20$, for example? Let's investigate this question for the case $p = .2$.

The basic idea, is that we should generate a large number of observations $x$ and from each one compute a 95% confidence interval for $p$. We then count up what proportion of those intervals include the true value of $p$ we used in the simulation. If the intervals were exact then that proportion would be very close to 0.95.

**根据中心极限定理，**当样本量 n 足够大时，二项分布会近似于正态分布。

```
n.rep <- 10000 ## number of CIs to compute
n.ci <- 0 ## counter for number that include the truth
n <- 20;p <- .2
for (i in 1:n.rep) {
  x <- rbinom(1,n,p) ## generate x
  p.hat <- x/n  ## compute the estimate of p
  sig.p <- sqrt(p.hat*(1-p.hat)/n) ## and its standard error
  if (p.hat - 1.96*sig.p <p && p.hat+1.96*sig.p > p) n.ci <- n.ci + 1 ## truth included?
}
n.ci/n.rep ## proportion of intervals including the truth
```

The result is about 0.92 — so when $n = 20$, 8% of intervals fail to include the true value, instead of the 5% that should. Many statistical methods rely on infinite sample size approximations, and this sort of simulation study is an essential way of checking out their performance at finite sample sizes.

```
n.rep <- 10000 ## number of CIs to compute
n.ci <- 0 ## counter for number that include the truth
n <- 20
p <- .2
x <- rbinom(n.rep, n, p) ## generate x
p.hat <- x / n ## compute the estimate of p
sig.p <- sqrt(p.hat * (1 - p.hat) / n) ## and its standard error
cp <- mean(p.hat - 1.96 * sig.p < p & p.hat + 1.96 * sig.p > p)
cp ## observed coverage probability
```

### Exercise

Write vectorized code to do the preceding calculation without a loop, and investigate how the coverage probability changes as you increase the sample size to 200, 2000 etc.

# 7 Reading and storing data in files

Since statistics is about understanding what data are telling us about the world, a statistical programmer needs to know how to read data into computer memory. Recall that computer memory (or 'random access memory') is the fast access temporary working memory of a computer. What is in it vanishes when you turn the computer off. It should not be confused with the long term storage provided by a hard disk, or solid state storage. The latter persists even when the computer is turned off, and is where 'computer files' are stored, organised in a hierarchy of folders/directories. Persistent storage is also sometimes accessed remotely over the internet, but we will start with reading in files stored locally on your computer.

## 7.1 working directories

R maintains a record of the current 'working directory'. This is the directory on your computer's file system where it will assume that a file is stored if you don't tell it otherwise. You can find out what the working directory currently is as follows.

```
> getwd()
[1] "/home/sw283" ## my working directory
```

The exact format depends on your operating system, but note that R uses `/` to separate subdirectories from parent directories, even on Windows. You can change the working directory as follows:

```
> setwd("/home/sw283/lnotes")
```

making sure that you specify a valid path for your filesystem. For example, on windows the path includes a letter identifying the disk or other storage device where the directory is located. e.g. `setwd("c:/foo/bar")` for subdirectory 'bar' of directory 'foo' on `c:`. Note the use of forward slashes in place of the windows default back slashes.

## 7.2 Reading in code: source()

Before reading in data, consider reading in code. The `source` function reads in code and runs it from a file. This is particularly useful for reading in function definitions from a file. For example:

```
source("mycode.r")
```

reads in the contents of `mycode.r`, first checking that it is valid R code and then running it. Running R code from a file like this is not exactly the same as typing it, or pasting it, into the R console. One main difference is that if you type the name of an object, and nothing else, at the console, then the object will be printed. If a line of sourced code contains the name of an object and nothing else, then the line causes nothing to happen. You would need to explicitly print the object if you want it to be printed. See `?source` for full details.

A useful function related to `source` is `dump`, which writes a text representation of some R objects to a file as R code, in a way that can later be read back in to R using `source`. For example suppose we have objects `a`, `alice` and Ba to save. All we have to do is supply `dump` with a text vector of the object names, and the name of the file to write them to:

```
dump(c("a","alice","Ba"),file="stuff.R")
```

## 7.3 Reading from and writing to text files of data

Text files have information stored in human readable form. You can open the file in an editor and understand it. Text files are a very portable way of storing data, but the files are quite a bit larger than is strictly necessary to store the information in the file. R has a number of functions for reading data from text files. All have many arguments to let you adapt to the way that the data has been stored.

As a simple example consider a file called `data.txt` containing

```
"name" "age" "height"
"sue"  21    1.6
"fred" 19    1.7
```

`scan` reads data from a file into a vector. For example:

```
> a <- scan("data.txt",what="c");a
Read 9 items
[1] "name" "age" "height" "sue" "21" "1.6"  "fred" "19" "1.7"
```

The type of the `what` argument tells `scan` what type of items it will be reading in. Character in this case, given `"c"` is of type character. `scan` treats white space as the separating items to read, but sometimes the separator is something else, for example a comma. In that case you can specify the separator using the `sep` argument. e.g. `sep=","`.

`scan` can also read in data in which different columns of data have different types, returning the different columns as different items of a list. In that case `what` is supplied as a list, with the type of the list items giving the types of the items to be read and stored. Here is an example in which the first line of the file is skipped using the `skip` argument.

```
> b <- scan("data.txt",what=list("a",1,1),skip=1); b
Read 2 records              第一列为字符，二三列为数值，跳过第一行
[[1]]
[1] "sue" "fred"

[[2]]
[1] 21 19

[[3]]
[1] 1.6 1.7
```

Actually, the file does not strictly need to have the data arranged in columns, simply in a repeated pattern.

Often the list form of the data is not as useful as having it in a data frame. So let's name the items in the list according to the first row of `data.txt` and convert to a data frame.

```
> names(b) <- scan("data.txt",what="c",n=3)
> d <- data.frame(b); d            扫描前三个项目为字符格式，存储在向量中
  name age height
1  sue  21    1.6
2 fred  19    1.7
```

Reading columns of data into a data frame in R is such a common task that a number of functions are provided for this purpose. The main one is `read.table`. Take a look at `?read.table` for the variations. An easier way of reading `data.txt` directly into `d` would be

```
d <- read.table("data.txt",header=TRUE)
```

where `header=TRUE` simply indicates that the first line gives the variable names. `sep` and `skip` can also be used with `read.table`. Unsurprisingly, the opposite task of writing a data frame out to a text file has its own function `write.table`. For example:

```
write.table(d,file="data1.txt")
```

On occasion it is necessary to read lines of text into a character vector, for further processing. The `readLines` function will do this, and there is an equivalent `writeLines` function as well.

To write data other than a data frame or character vector to a text file you can use the `cat` or `write` functions: both take a `file` argument specifying the file name to write to. `dump` is another option covered in the previous section.

### Exercise

Create a version of text file `data.txt` outside R yourself, and read it into R as described above.

## 7.4 Reading and writing binary files

To save large R objects or data frames to file, it is more efficient to use a binary format and compression. `save` can do this. It takes a sequence of R objects, and writes them to the file specified in its `file` argument. `load` will read the objects back in from the file. Here is a simple example:

```
A <- matrix(1:6,3,2); y <- rnorm(100) ## create some objects
save(A,y,file="stuff.Rd") ## save them to a binary file
rm(y,A)      ## delete the original objects
load("stuff.Rd") ## load them back in again from the file
```

`save` does allow you to choose to save the objects as text as well, but this requires larger files: for the above example twice as large, but sometimes *much* larger.

## 7.5 Reading data from other sources

Rather than reading standard files from storage on your local computer, you might want to read data from a file subject to some standard compression algorithm, or over a local network, or from a URL. Most of the functions for reading and writing data can use one of these other types of *connection* as their file argument. See `?connection` for details. Here is a single common example of reading data from a web address

```
raw <- readLines("https://michaelgastner.com/DAVisR_data/homicides.txt")
```

in this case lines of data are read into a character vector.

# 8 Data re-arrangement and tidy data

In raw form many data sets are messy and have to be tidied up for statistical analysis. By tidying is meant that the data are re-arranged conveniently for analysis, not that the information in the data is in anyway modified unless an actual error is identified (no removal of outliers, for example). Since 'data tidying' or 'data re-arrangement' sound too boring and clerical for many enthusiasts for Modern Data Science, you will often hear terms like 'data wrangling' and 'data munging[10]' used instead. These sound suitably rugged, as if the types engaged in them return home aching and covered in dust and mud after a day engaged in the honest toil of these muscular pursuits.

By tidy data is meant that data are arranged in a matrix so that:

1. Each column is a variable.
2. Each row is an observation (a unique combination of variables).

and it is recognised that what counts as 'an observation' can depend on the subset of variables of interest. For example, this data frame is in tidy form if interest is in student's grades each week.

```
name    d.o.b.  nationality week grade
George  11/2/01 GB          1    B
George  11/2/01 GB          2    C
```

[10] a term that started out as the derogatory 'mash until no good'.

```
George  11/2/01 GB          3    A
Anna    15/6/00 GB          1    A
Anna    15/6/00 GB          2    B
Anna    15/6/00 GB          3    A
.       .       .           .    .
```

But if we are only interested in birth dates[11] and nationality of students, then we would really want to simplify to

```
name    d.o.b.  nationality
George  11/2/01 GB
Anna    15/6/00 GB
.       .       .
```

It is easy to get the second data frame from the first (`marks`, say) in base R using

```
student <- unique(marks[,c("name","d.o.b.","nationality")])
```

which has selected the named columns of interest, and then found the unique rows. If you want to refer each row in `marks` to its row in `student` then you can use

```
library(mgcv)
student <- uniquecombs(marks[,c("name","d.o.b.","nationality")])
ind <- attr(student,"index") (1, 1, 1, 2, 2, 2)
## ... ind[i] is row of 'student' corresponding to row i of 'marks'
```

At this point, you are hopefully wondering about replicate data, and the definition of 'observation' as 'unique combination of variables'? If I repeat an experiment under identical conditions, could I not occasionally get identical results, and hence two identical rows in a data frame that are different observations? Well yes, but only if you do not include a variable recording the 'replicate number'. If you don't record the replicate number then you do not really have tidy data, as it is not possible to distinguish the common error of accidental replication of some recorded data, from genuine replicates.

The R add on packages collectively known as the *tidyverse* provide a large number of functions for data tidying – we won't cover these here, as it takes us too far from fundamentals of statistical programming: but see the *Research Skills* course next semester.

## 8.1 Baltimore homicides: data tidying and regular expressions

Let's look at re-arranging a fairly messy data set on homicides in Baltimore. The raw data can be obtained using

```
raw <- readLines("https://michaelgastner.com/DAVisR_data/homicides.txt")
raw[1] ## examine first record
[1] "39.311024, -76.674227, iconHomicideShooting, 'p2', '<dl><dt>Leon Nelson</dt>
<dd class=\"address\">3400 Clifton Ave.<br />Baltimore, MD 21216</dd><dd>black male,
 17 years old</dd><dd>Found on January 1, 2007</dd><dd>Victim died at Shock Trauma
 </dd><dd>Cause: shooting</dd></dl>'"
```

The subsequent entries in `raw` look similar, but they are not completely standardized, as we will see. For analysis, we need to extract the information that has been read into `raw` in standardized form. For example, we might want to extract the longitude and latitude location information (the first 2 numbers), the victim name, the race information (black, white, hispanic etc), age etc. To do this requires some effort, first trawling through the data to check how the information is stored, and whether it is consistent across records.

To start, consider a relatively easy task - the location information. This is always first, with latitude and longitude separated by a comma. So we could simply split up each record wherever there is a comma, and read off the first two elements from each resulting split record. `strsplit` can be used to do this, as follows:

```
rl <- strsplit(raw,",")
lat <- as.numeric(sapply(rl,function(x) x[1]))
lon <- as.numeric(sapply(rl,function(x) x[2]))
```

[11] calender dates often need handling in special ways — see `?as.Date` and `?julian` to get started.

- `strsplit(raw,",")` splits each character string in text vector `raw` into a vector of character strings, with the splits occurring wherever there is a comma (the commas are dropped). `strsplit` returns a list of character vectors, with an element for each element of `raw`: this is stored in `rl`.
- Now we just need to extract the first and second element of each character vector in `rl`. `sapply` applies a function to each element of a list, returning the result as a vector. So `sapply(rl,function(x) x[1])` applies a function that simply extracts the first element of a vector to `rl`. The result is a vector of latitudes, as text, which are then converted to numeric and stored in `lat`.
- Longitudes are extracted in the same way.

Extracting names is not so easy. A consistent pattern is that the name record starts `<dt>` and ends `</dt>`, although the record may also include a web address pointing to a further record. I therefore first tried to split on the occurrence of `dt`, but even after trying to clean out the web addresses one of the resulting names was 86 characters long, and still contained a web address. The problem turned out to be that the victims names was Nancy Schmi**dt**.

So it is better to use `<dt>` and `</dt>` in some way. Indeed if you look at record 1 above, you'll see that I ought to be able to snip out the name very neatly using these. However the data turn out to be messier than that. e.g.

```
> raw[300]
[1] "39.28957200000, -76.57039400000, icon_homicide_shooting, 'p489', '<dl><dt><a
href=\"http://essentials.baltimoresun.com/micro_sun/homicides/victim/489/alton-alston
\">Alton Alston</a></dt><dd class=\"address\">200 S. Clinton St.<br />Baltimore,
 MD 21224</dd><dd>Race: Unknown<br />Gender: male<br />Age: 18 years old</dd><dd>
 Found on December 1, 2008</dd><dd>Victim died at Unknown</dd><dd>Cause: Shooting
 </dd><dd><a href=\"http://www.baltimoresun.com/news/local/baltimore_city/bal-
 eastside1201,0,5413416.story\">Read the article</a></dd></dl>'"
```

The web address in the name record needs stripping out. This sort of problem needs a bit more than the simple splitting on a pattern met so far: it needs *regular expressions*.

### 8.1.1 Regular expressions

Regular expressions provide a way of matching patterns in text that are more flexibly defined than simple fixed strings. For example, suppose I wanted to find all words in a which at some point an 'i' occurs two characters after a 'c' (as in 'clip' or 'script'), or I want to find all words starting with 'f' and ending with 'k' (as in 'fork' or 'flunk'). We can accomplish such tasks by constructing strings which contain special characters interpreted as matching particular patterns. Functions such as `grep` and `gsub` can understand these *regular expressions* (unless we turn off this ability, with argument `fixed=TRUE`). You can get a full description of the available regular expressions by typing `?regex` in R. Note the slight irritation that control characters referred to as e.g. `\b` in the help file actually have to be used as `\\b` in regular expressions.

Within regular expressions the characters `. \ | ( ) [ { ^ $ * + ?` have special meanings. If you want them treated as regular characters instead, you have to precede them by `\\` (or if you don't need any of them to have their special meaning, use argument `fixed=TRUE`). Here we will use a couple of these: '`.`' can be used to represent any single character. This could accomplish the first task above:

```
> txt <-
"He scribbled a note on his clipboard and clicked his heels as he clinched the deal."
> grep("c.i",strsplit(txt," ")[[1]])
[1] 7 9 14
```

For the second example we want to allow an arbitrary number of letters between the letters at the start and end of the word. '`?`' '`*`' and '`+`' all modify the character preceding them: '`?`' indicates that it's optional, and will be matched at most once; '`*`' indicates that it should be matched zero of more times, and '`+`' that it should be matched 1 or more times. We want to match any number of characters between 'f' and 'k', and '`.*`' is how to do that:

```
> txt1 <-
"To flunk this exam would be disaster, thought Phil, picking up his fork to eat "
> grep("f.*k",strsplit(txt1," ")[[1]])
[1] 2 13
```

But actually this is not general enough. Consider

```
> txt2 <-
"Flunk this exam and I'll be toast, thought Farouk, forking beans into his mouth."
> grep("f.*k",strsplit(txt2," ")[[1]])
[1] 10
```

...the wrong answer - the two words we wanted have been missed, and 'forking' has been wrongly matched. Consulting `?regex` again, it turns out that we can specify sets of characters to match, by enclosing them in square brackets, for example `[Ff]`. We can also specify the end of a word by `\\b`. Putting these together:

```
> grep("\\b[Ff].*k\\b",strsplit(txt2," ")[[1]])
[1] 1 9 ## success
```

**Back to the homicide data**

With regular expressions it is now possible to extract the names automatically, as follows

```
get.name <- function(x) {
  x1 <- gsub(".*<dt>","",x) ## strip out everything to <dt>
  x1 <- gsub("</dt>.*","",x1) ## strip out everything after </dt>
  x1 <- gsub("<a.*\">","",x1) ## strip out any web addresses
  sub("</.*","",x1) ## strip out any further tags
}
name <- sapply(raw,get.name,USE.NAMES=FALSE)
```

1. Notice how I have broken the task down into parts in `get.name`, with each intermediate result stored before it is updated. This 'defensive coding' facilitates debugging if something goes wrong. I can then use a debugger to step through `get.name` line by line, checking that each intermediate step is what I expected. If I had written the whole operation as a set of nested function calls, it would have been much harder to debug. Styles of programming that involve 'piping' the output of one function directly to the input of another, suffer from the same code maintenance and debugging problems.

2. `sapply` applies the functions specified as its second argument to each element of its first argument, returning the result in a vector. The `USE.NAMES=FALSE` argument avoids `sapply` attaching names to the vector it creates: by default it uses the entire original string in `raw[i]` as the label for `name[i]`. This would be inconvenient.

Finally let's extract race. More data inspection reveals that the information comes immediately after the first `<dd>` tag, but there are two forms: either the race qualifier is the first word in a longer phrase, or it comes after the text 'Race:'. So one way of extracting something appropriate is

raw **是字符向量：**(' ', ' ', ...)
rl **是列表：**[( , , ), ( , , ), ...]

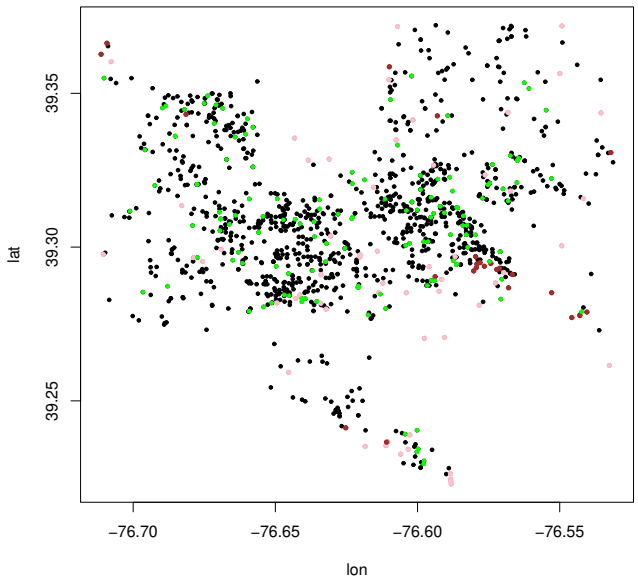
```
rl <- strsplit(raw,"<dd>") ## split so race info second element of each record
race <- sapply(rl,function(x) sub("<.*","",x[2]) ) ## strip trailing tags
ii <- grep("Race:",race) ## locate 'Race:' cases
race[ii] <- gsub("Race: ","",race[ii]) ## extract race info for 'Race:' cases
race[-ii] <- gsub(" .*","",race[-ii]) ## drop rest of text in other cases
```

Putting the information extracted into a data frame gives

```
> dat <- data.frame(lon=lon,lat=lat,name=name,race=tolower(race))
> head(dat)
        lon      lat               name  race
1 -76.67423 39.31102        Leon Nelson black
2 -76.69895 39.31264         Eddie Golf black
3 -76.64988 39.30978   Nelsene Burnette black
.    .        .                      .    .
```

We might immediately want to plot the locations of the homicides, coded by race. The following does this for main racial groups of black (64% of population of Baltimore), white (28%) and hispanic (4%), other races and unknown are lumped together (the vast majority of these are unknown or not recorded).

```
with(dat,plot(lon,lat,pch=19,cex=.5))
ii <- dat$race=="white"
points(dat$lon[ii],dat$lat[ii],col="pink",pch=19,cex=.6)
ii <- dat$race=="hispanic"
points(dat$lon[ii],dat$lat[ii],col="brown",pch=19,cex=.6)
ii <- !(dat$race=="black"|dat$race=="white"|dat$race=="hispanic")
points(dat$lon[ii],dat$lat[ii],pch=19,cex=.5,col="green")
```




```
grep("<dd>Cause:", raw) -> ii
## check for cause field

raw[-ii]
## OK these really are missing - code as NA

x <- gsub(".*<dd>Cause: ", "", raw)
## cut text before actaul cause record

cause <- gsub("</dd>.*", "", x)
## cut text from end of record

cause[-ii] <- NA
## set missings to NA

mean(tolower(cause) == "shooting", na.rm =
TRUE)
## prop shooting
```

**Exercise**

Write code to extract the cause of death. In this case you'll see that most records have a cause field starting `<dd>Cause:`. Use `grep` to create an index of the entries in `raw` containing this sequence, and hence check the entries that do not have this field to see if cause is coded differently there, or is simply missing. Then write code to extract the cause data, doing something sensible about the records without the `<dd>Cause:` entry. Find the proportion of homicides that are by shooting.

# 9 Statistical Modelling: linear models

统计建模的目的是通过建立模型来模拟数据生成的过程，以帮助我们了解数据背后的人群特征或系统的行为

So far we have looked at programming data manipulation tasks, and at some simulation from stochastic models. What about statistical modelling? That is the process of using models of the data generating process to help us learn about the population from which the data were generated (or, if you prefer, the *system* that generated the data). *Linear models* are a good place to start, since such models introduce many of the concepts used in statistical modelling more generally.

The general setup is that we have observations of a random *response* variable, $y_i$, accompanied by (fixed or random) *predictor* variables (a.k.a. *covariates*), $x_{ij}$. Here observations are indexed by $i = 1, \ldots, n$ and predictor variables by $j = 1, \ldots, p$, and $p \leq n$. Predictor variables come in two basic varieties:

度量变量
1. *Metric variables* are numerical measurements of something. e.g. height, weight, litres of fuel used per 100km, temperature etc.
因子变量
2. *Factor variables* are labels indicating membership of some group (the groups are known as *levels* of the factor, although they are not treated as having any particular ordering). For example, nationality, sex, variety of wheat, species of bird. See also section 5.2.4.

Obviously on occasion factors and metric variables may play rather similar roles - for example when we categorize people into age groups.

Considering all metric predictors first, the generic linear model structure is

$$y_i = \beta_0 + x_{i1}\beta_1 + x_{i2}\beta_2 + \cdots + x_{ip}\beta_p + \epsilon_i$$

where the $\beta_j$ are unknown parameters to be estimated from data, and the $\epsilon_i$ are independent random variables, with zero expected value, and variance $\sigma^2$. For some purposes, we will also assume that they are normally distributed. Notice that if we take expectations of both sides of the model we get $E(y_i) = \beta_0 + x_{i1}\beta_1 + x_{i2}\beta_2 + \cdots + x_{ip}\beta_p$.

The model is *linear* in the parameters, $\beta_j$, and noise, $\epsilon_i$. It can be non-linear in the predictors. For example:

$$y_i = \beta_0 + x_{i1}\beta_1 + x_{i1}^2\beta_2 + x_{i2}\beta_3 + x_{i1}x_{i2}\beta_4 \cdots + \epsilon_i$$

In a way it is obvious that this must be possible: we put no restrictions on the predictors, so $x_{i1}^2$ or $x_{i1}x_{i2}$ must be as valid as predictors as $x_{i1}$ and $x_{i2}$ themselves.

This observation leads us on to linear models involving factor variables. The key idea is best seen by example. Suppose $x_{i1}$ is a factor variable that can take values `a`, `b` or `c`. The idea is then that a parameter will be added to the model depending on which level the factor takes. Mathematically we have a model something like:

$$y_i = \beta_0 + \mathbb{I}(x_{i1} = \mathtt{a})\beta_1 + \mathbb{I}(x_{i1} = \mathtt{b})\beta_2 + \mathbb{I}(x_{i1} = \mathtt{c})\beta_3 + x_{i2}\beta_4 + \cdots + \epsilon_i$$

where $\mathbb{I}()$ is the indicator function, taking the value 1 when its argument is true and 0 otherwise. There is a catch, however. In the preceding model we could add any constant $k$ to $\beta_0$ while subtracting the same constant from $\beta_1$, $\beta_2$ and $\beta_3$ and the numerical values produced by the right hand side of the model would be unchanged. Clearly the model has redundancy - an infinite set of parameter values can produce any given value of the right hand side. Hence the model parameters can not be uniquely estimated from data: the model is not *identifiable*.

Happily, this lack of identifiability can be rectified by simply dropping one of the parameters from the redundant set. We could drop $\beta_0$, but then we get the same problem again is we add a second factor variable to the model, so a better option is to drop the parameter associated with the first level of each factor. This is what R does automatically. In the preceding case we then get:

$$y_i = \beta_0 + \mathbb{I}(x_{i1} = \mathtt{b})\beta_1 + \mathbb{I}(x_{i1} = \mathtt{c})\beta_2 + x_{i2}\beta_3 + \cdots + \epsilon_i \tag{1}$$

(note the closing up of $\beta$ indices). The new version can predict anything that the old version could, but the redundancy/lack of identifiability has gone. The only price paid is that the interpretation of $\beta_0$ and the parameters for the factor effect now changes: $\beta_0$ is now the 'intercept' parameter that applies when $x_{i1}$ has value `a`. $\beta_1$ and $\beta_2$ are now the *differences* between the effect of level `b` and level `a` and between level `c` and level `a`, respectively.

Note that writing models out with indicator functions can get very clumsy, especially for factors with many levels, so alternative ways of expressing the same model can be useful. One approach uses multiple indices, with an index for each level of each factor variable in the model. For example

$$y_{ij} = \beta_0 + \gamma_j + z_{ij}\beta_1 + \epsilon_{ij} \text{ if } x_{ij} \text{ takes its } j^{\text{th}} \text{ level}$$

is a structure exactly like that of (1), but to express it clearly, multiple Greek letters have been used for parameters, and multiple Roman letters for variables. Identifiability in this case is imposed with a restriction, such as $\gamma_1 = 0$. Multiple indices like this can also get messy, so it can be more elegant to deal with factors by simply indexing the associated parameters by levels of a factor. For example if $x_i$ is a factor (perhaps taking levels `a`, `b` or `c`) then a model might be expressed as

$$y_i = \beta_0 + \gamma_{x_i} + z_i\beta_1 + \epsilon_i$$

...the idea being that there is a different $\gamma$ parameter for each level of the factor (e.g. $\gamma_{\mathtt{a}}$, $\gamma_{\mathtt{b}}$, $\gamma_{\mathtt{c}}$, or whatever). Again identifiability constraints are written as restrictions on the parameters, such as $\gamma_{\mathtt{a}} = 0$.

The point here is to recognise that there are different ways to express exactly the same model structure - which is most convenient depends on the context.

## 9.1 Statistical interactions

Clearly linear models allow us to include the effects of several predictors of mixed type (metric and/or factor) in an additive way — with the effects of the variables simply added to each other. But what about the case in which the effects of variables do not simply add up, but rather the effect of one variable is to modify the effect of another variable. In statistical modelling this is known as an **interaction**. It is one of those simple key concepts that it is essential to understand, so read the following very carefully, even if you already know about interactions.

What does *the effect of one predictor modifies the effect of another predictor* mean in concrete model terms? Consider an example. Suppose a model contains the effect of predictor $x_i$ in a simple linear way:

$$y_i = \beta_0 + x_i\beta_1 + \epsilon_i$$

i.e. $E(y_i)$ has a straight line relationship to $x_i$, with slope parameter $\beta_1$. But now suppose that the parameters of this straight line relationship themselves change according to a second predictor variable, $z_i$. How? Perhaps the effect of $z_i$ is also linear, so that $\beta_0 = \gamma_0 + z_i\gamma_1$ and $\beta_1 = \gamma_2 + z_i\gamma_3$. Substituting into the original model we get

$$y_i = \gamma_0 + z_i\gamma_1 + x_i\gamma_2 + x_iz_i\gamma_3 + \epsilon_i$$

We are not limited to producing interactions of effects that are linear in the covariates. Interactions of any sort of effect can be produced in the same way. Suppose, for example, that the effect of $x_i$ is quadratic so that

$$y_i = \beta_0 + x_i\beta_1 + x_i^2\beta_2 + \epsilon_i$$

and that this effect should vary linearly with $z_i$. Then $\beta_0$ and $\beta_1$ are as before, while $\beta_2 = \gamma_4 + z_i\gamma_5$, so

$$y_i = \gamma_0 + z_i\gamma_1 + x_i\gamma_2 + x_iz_i\gamma_3 + x_i^2\gamma_4 + x_i^2z_i\gamma_5 + \epsilon_i.$$

Notice how in both the preceding examples the model includes some terms that would have been there if the effects had been purely additive: these are known as the **main effects**, while the extra terms are referred to as **interaction terms**. For the last model the main effects are $z_i\gamma_1$ and $x_i\gamma_2 + x_i^2\gamma_4$, while the interaction term is $x_iz_i\gamma_3 + x_i^2z_i\gamma_5$.

Exactly the same idea applies to factor variables. For example we might want the effects of $x_{i1}$ and $x_{i2}$ from model (1) to interact. Whether we allow the parameters for the $x_{i1}$ effect to change linearly with $x_{i2}$, just like the main effect of $x_{i2}$, or we allow the parameter for the $x_{i2}$ effect to depend on $x_{i1}$ according to the $x_{i1}$ model, we end up with the same model...

$$y_i = \beta_0 + \mathbb{I}(x_{i1} = \texttt{b})\beta_1 + \mathbb{I}(x_{i1} = \texttt{c})\beta_2 + x_{i2}\beta_3 + \mathbb{I}(x_{i1} = \texttt{b})x_{i2}\beta_4 + \mathbb{I}(x_{i1} = \texttt{c})x_{i2}\beta_5 + \cdots + \epsilon_i$$

The pattern is similar for interactions between factor variable effects. For the above example, suppose $x_{i2}$ was actually a factor variable, with levels `g1` and `g2`. The first level, `g1`, would get dropped of course, to ensure identifiability. Allowing the parameters for the $x_{i1}$ main effect to vary according to the $x_{i2}$ main effect (or vice-versa) we get (ignoring any other model components)

$$y_i = \beta_0 + \mathbb{I}(x_{i1} = \texttt{b})\beta_1 + \mathbb{I}(x_{i1} = \texttt{c})\beta_2 + \mathbb{I}(x_{i2} = \texttt{g2})\beta_3 + \mathbb{I}(x_{i1} = \texttt{b})\mathbb{I}(x_{i2} = \texttt{g2})\beta_4 + \mathbb{I}(x_{i1} = \texttt{c})\mathbb{I}(x_{i2} = \texttt{g2})\beta_5 + \epsilon_i$$

Given the model identifiability problems when first introducing factors, is it necessary to worry about identifiability in interactions? Not if the main effects are made identifiable first, and the interaction terms are constructed from these identifiable main effects: the identifiability is 'inherited' from the main effects.

Pay careful attention to what a statistical interaction is not. It *is not* about the relationship of one predictor to another. It *is* about their combined effect on the response.

## 9.2 Computing with linear models: model matrices and model formulae

To compute with linear models it is very helpful to write the model in the general matrix vector form

$$\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}$$

where $\mathbf{y}$ is an $n$ vector of response variable values, $\mathbf{X}$, the **model matrix**, is an $n \times p$ matrix determined by the model structure and predictor variables, $\boldsymbol{\beta}$ is a parameter vector and $\boldsymbol{\epsilon}$ is a vector of independent zero mean,

constant variance random variables (possibly normally distributed). How do the models we have seen so far fit into this form?

This is easiest to see by example, so consider model (1) again, and assume it only has the 4 parameters shown. As written there is an index $i$ which runs from 1 to $n$, so the model is really the system of equations

$$\begin{aligned}
y_1 &= \beta_0 + \mathbb{I}(x_{11} = \mathtt{b})\beta_1 + \mathbb{I}(x_{11} = \mathtt{c})\beta_2 + x_{12}\beta_3 + \epsilon_1 \\
y_2 &= \beta_0 + \mathbb{I}(x_{21} = \mathtt{b})\beta_1 + \mathbb{I}(x_{21} = \mathtt{c})\beta_2 + x_{22}\beta_3 + \epsilon_2 \\
&\cdot \quad \cdot \qquad \cdot \\
&\cdot \quad \cdot \qquad \cdot \\
y_n &= \beta_0 + \mathbb{I}(x_{n1} = \mathtt{b})\beta_1 + \mathbb{I}(x_{n1} = \mathtt{c})\beta_2 + x_{n2}\beta_3 + \epsilon_n
\end{aligned}$$

Obviously this can be written as

$$\begin{bmatrix} y_1 \\ y_2 \\ \cdot \\ \cdot \\ y_n \end{bmatrix} = \begin{bmatrix} 1 & \mathbb{I}(x_{11} = \mathtt{b}) & \mathbb{I}(x_{11} = \mathtt{c}) & x_{12} \\ 1 & \mathbb{I}(x_{21} = \mathtt{b}) & \mathbb{I}(x_{21} = \mathtt{c}) & x_{22} \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ 1 & \mathbb{I}(x_{n1} = \mathtt{b}) & \mathbb{I}(x_{n1} = \mathtt{c}) & x_{n2} \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \beta_3 \end{bmatrix} + \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \cdot \\ \cdot \\ \epsilon_n \end{bmatrix},$$

which is clearly in the form $\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}$.[12] To make this more concrete still, suppose that $x_{\cdot 1} = (\mathtt{c}, \mathtt{b}, \mathtt{b}, \mathtt{c}, \ldots, \mathtt{b})^{\mathrm{T}}$, and $x_{\cdot 2} = (0.3, -2, 27, 3.4, \ldots, 10)^{\mathrm{T}}$. Then the model is

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ \cdot \\ \cdot \\ y_n \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 0.3 \\ 1 & 1 & 0 & -2 \\ 1 & 1 & 0 & 27 \\ 1 & 0 & 1 & 3.4 \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ 1 & 1 & 0 & 10 \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \beta_3 \end{bmatrix} + \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \epsilon_3 \\ \epsilon_4 \\ \cdot \\ \cdot \\ \epsilon_n \end{bmatrix}.$$

To understand linear models it is essential to understand how model matrices relate to the models as they are usually written down, especially how factor variables expand to several columns of binary indicators. The other key concepts to understand to the point that they are second nature are: factors, identifiability and interactions.

So how do we actually compute with these modelling concepts? R has functionality designed to make the setting up of model matrices very straightforward, based on the notion of a **model formula**. An R model formula is a compact description of a model structure in terms of the names of the variables in the model. For example:

```
y ~ x + z
```

specifies a model $y_i = \beta_0 + x_i\beta_1 + z_i\beta_2 + \epsilon_i$, if $x$ and $z$ are metric variables (an intercept is included by default). Whatever is to the left of `~` is the response variable, while what is to the right specifies how the model depends on predictor variables. `x + z` implies that we want a main effect of `x` and a main effect of `z`. The exact form of these main effects depends on whether each predictor is a factor or a metric variable. Notice how `+` has a special meaning in the formula. It basically means *and*. It does not mean addition.

Operators `+`, `-`, `^`, `*` and `:` all have special meanings when used in a formula, *except* when they are using in arguments of a function inside a formula. So while `y ~ x + z` indicates to include effects of `x` and of `z`, not include and effect of $x + z$, the formula

```
y ~ log(x + z)
```

does imply the addition of $x$ and $z$ before the log is taken. This means that it is easy to restore the usual arithmetic meaning of operators, by including the terms as arguments of the identity function `I()` in model formulae. For example, to implement $y_i = \beta_0 + x_i\beta_1 + x_i^2\beta_2 + \epsilon_i$, use

```
y ~ x + I(x^2)
```

[12] Obviously we do not require that every linear model we write down has parameters called $\beta_0, \beta_1, \beta_2$ etc. The point is simply that whatever parameters the model has, they can be stacked up into one parameter vector, here called $\boldsymbol{\beta}$.

Model formula operators `*` and `:` implement interactions. `a*b` indicates that the main effect terms and the terms for the interaction of `a` and `b` are to be included in a model. `a:b` just includes the interaction terms, but not the main effects. So `a*b` and `a + b + a:b` are equivalent. Higher order interaction operate the same way. For example `a*b*c` is equivalent to `a + b + c + a:b + a:c + b:c + a:b:c`

Of course you might want to include interactions of several variables, only up to a certain order. This is where the `^` operator comes in. Suppose we want the interactions of `a`, `b`, `c` and `d` up to order 2 (and not up to order 4 as `a*b*c*d` would generate). `(a+b+c+d)^2` will do just that, being equivalent to

```
a + b + c + d + a:b + a:c + a:d + b:c + b:d + c:d
```

Similarly `(a+b+c+d)^3` will generate all interactions up to order 3. Notice how there is no interaction of a variable with itself. You could argue that there should be, but actually if you try and make a factor variable interact with itself you get back the original factor, so it is cleaner to simply define metric self-interactions in the same way, which is what is done.

It is also helpful to be able to specify that some terms otherwise included should be excluded and the − operator allows this to happen. For example, suppose be wanted to exclude the last two interactions in the previous example. `(a+b+c+d)^2-c:d-b:d` would do this. But actually the most common use of − is to exclude the intercept term, by including '`-1`' in the formula.

R function `model.matrix` takes a model formula and a data frame containing the variables referred to by the formula, and constructs the specified model matrix. Here it is in action. First consider the `PlantGrowth` data frame in R, which contains `weights` of plants grown under 3 alternative treatment `groups`. Suppose we want a simple linear model in which `weight` depends on `group`.

```
> X <- model.matrix(weight~group,PlantGrowth)
> X
   (Intercept) grouptrt1 grouptrt2
1            1         0         0
2            1         0         0
.            .         .         .
10           1         0         0
11           1         1         0
12           1         1         0
.            .         .         .
20           1         1         0
21           1         0         1
22           1         0         1
.            .         .         .
```

As a further example, consider the `mpg` data frame provided with `ggplot2`. Actually `mpg` is provided as a modified data frame given the obscure name of a 'tibble'[13]. The features of a tibble are not any help here, so we may as well convert back to a regular data frame. Then let's consider a linear model which attempts to model fuel efficiency (`cty` miles per US gallon) using predictors `trans` and `displ`. `trans` is the type of transmission - manual or automatic + other information such as number of gears. For now let's strip out the other information and just consider the two levels . `displ` is total cylinder volume in litres.

```
> mpg1 <- data.frame(mpg) ## convert to regular data frame
> head(mpg1) ## fl is fuel, but levels undocumented, so unusable
  manufacturer model displ year cyl      trans drv cty hwy fl   class
1         audi    a4   1.8 1999   4   auto(l5)   f  18  29  p compact
2         audi    a4   1.8 1999   4 manual(m5)   f  21  29  p compact
3         audi    a4   2.0 2008   4 manual(m6)   f  20  31  p compact
4         audi    a4   2.0 2008   4   auto(av)   f  21  30  p compact
5         audi    a4   2.8 1999   6   auto(l5)   f  16  26  p compact
6         audi    a4   2.8 1999   6 manual(m5)   f  18  26  p compact
> mpg1$trans <- factor(gsub("\\(.*\\)","",mpg1$trans)) ## convert to 2 level
> head(model.matrix(cty~trans+displ,mpg1)) ## get model matrix
```

[13] the documentation for tibbles says that they help with coding expressively, how inventing new meaningless names helps that is unclear.

```
  (Intercept) transmanual displ
1           1            0   1.8
2           1            1   1.8
3           1            1   2.0
4           1            0   2.0
5           1            0   2.8
6           1            1   2.8
```

...make sure that this is what you were expecting!

As another example, suppose we want an interaction between `displ` and `trans`

```
> head(model.matrix(cty~trans*displ,mpg1))
  (Intercept) transmanual displ transmanual:displ
1           1            0   1.8              0.0
2           1            1   1.8              1.8
3           1            1   2.0              2.0
4           1            0   2.0              0.0
5           1            0   2.8              0.0
6           1            1   2.8              2.8
```

...make sure you can interpret this model. There is a straight line relationship between `cty` and `displ` for automatic transmission cars, given by columns 1 and 3. There is then a second straight line dependence on `displ` for the difference in city miles per gallon between manual and automatic cars.

## Exercises

1. Suppose you have a response variable $y_i$ a metric predictor variable $\mathbf{x} = (0.1, 1.3, 0.4, 1.4, 2.0, 1.6)^{\mathrm{T}}$ and a factor variable $\mathbf{z} = (\mathtt{a}, \mathtt{a}, \mathtt{fred}, \mathtt{a}, \mathtt{c}, \mathtt{c})$. Write out the model matrix for the model $y_i = \beta_0 + x_i\beta_1 + \gamma_{z_i}$, making sure that it is identifiable.
2. Consider factor variables $\mathbf{x} = (\mathtt{a}, \mathtt{a}, \mathtt{b}, \mathtt{a}, \mathtt{b}, \mathtt{a})^{\mathrm{T}}$ and $\mathbf{z} = (\mathtt{ctrl}, \mathtt{trt}, \mathtt{trt}, \mathtt{trt}, \mathtt{ctrl}, \mathtt{ctrl})^{\mathrm{T}}$. Write out an identifiable model matrix for the linear model which includes main effects and an interaction of $\mathbf{x}$ and $\mathbf{z}$.
3. Now suppose that $\mathbf{x}$ is as in the previous exercise, but $\mathbf{z} = (1, 2, 1, 3, 4, 2)$. Write out the model matrix for a model containing main effects of $\mathbf{x}$ and $\mathbf{z}$ and their interaction.

## 9.3 Fitting linear models

The parameters of linear models are estimated by *least squares*. Defining $\mu_i = E(y_i)$ (so vector $\boldsymbol{\mu} = \mathbf{X}\boldsymbol{\beta}$), we seek parameter estimates $\hat{\boldsymbol{\beta}}$ that minimise

$$\sum_{i=1}^{n}(y_i - \mu_i)^2 = \|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\|^2.$$

How that is done mathematically will be covered as an example of matrix computation. For the moment, consider the built in R functions for linear modelling. The main function is `lm`. Here it is in use fitting a basic model for `cty` miles per gallon, in which $E(\mathtt{cty})$ depends linearly on `trans` and `displ`.

```
> m1 <- lm(cty ~ trans + displ, mpg1)
> m1

Call:
lm(formula = cty ~ trans + displ, data = mpg1)

Coefficients:
(Intercept) transmanual displ
   25.4609    0.7842   -2.5520
```

`lm` takes a model formula, sets up the model matrix and estimates the parameters. Typing the names of the returned model object, causes it to be printed, giving the `lm` function call and reporting the fitted parameter values. Notice how the parameters are identified by the name of the predictor variable they relate to. For factor variables, where there may be several parameters, parameters are referred to by the name of the factor, and the relevant level.

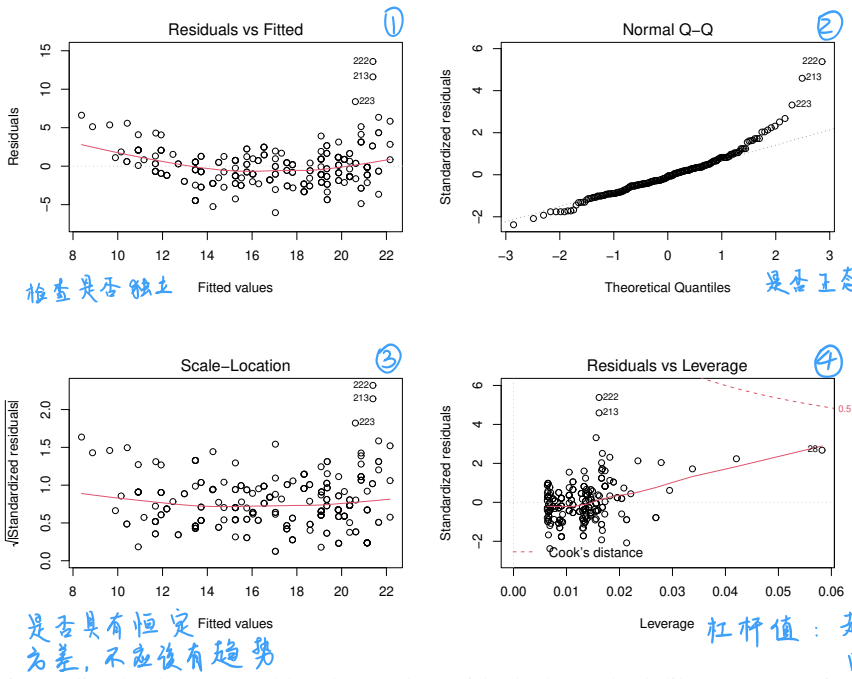
The first thing to do with a linear model is to check its assumptions. These are essentially that the $\epsilon_i$ are independent with zero mean and constant variance, and less importantly, that they are normally distributed. The usual way to check this is with residual plots. The *residuals* are define as

$$\hat{\epsilon}_i = y_i - \hat{\mu}_i$$

where $\hat{\boldsymbol{\mu}} = \mathbf{X}\hat{\boldsymbol{\beta}}$ - they are effectively the estimates of the $\epsilon_i$ . They are plotted in ways that should show up problems with the assumptions. In fact simply calling the `plot` function with a linear model object as argument generates four default residual plots.

用于评估线性模型的假设是否成立

```
> plot(m1)
```




① 检查是否独立

② 是否正态分布

③ 是否具有恒定方差，不应该有趋势

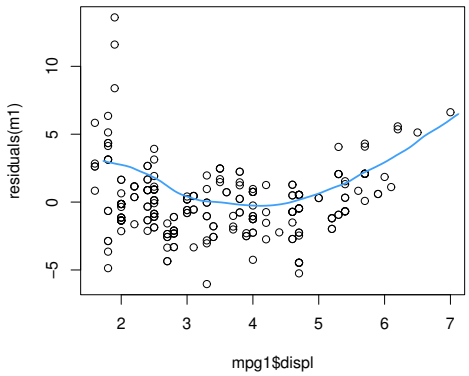
④ 杠杆值：越大意味着该观测点对回归模型的影响越大，远离主要群体

The top left plot immediately shows a problem here - the residuals do not look like zero mean independent random deviates. There is a clear pattern in the mean of the residuals (the red curve is a running smoother, helping to visualize that trend). Interpreting the remaining plots is pointless given this problem, but here is what they show, anyway. The top right plot is a normal quantile-quantile plot - sorted residuals are plotted against quantiles of a standard normal distribution: close to a straight line is expected for normal data (but normality is not the most important assumption). The lower left plot standardizes the residuals so that any trend in the mean indicates violation of the constant variance assumption. The final plot indicates whether some points have a combination of high residual and unusual predictor values that makes them particularly influential on the whole model fit — see the generalized regression models course for more on these.

What might cause the systematic pattern in the residuals? One possibility is that the model for the relationship between `cty` and `displ` is wrong. To check, plot the residuals against displacement.

```
> plot(mpg1$displ,residuals(m1))
```

displ 和残差之间的关系




表明当前模型未正确捕捉 displ 和 cty 之间的关系，它们之间可能是非线性的。

...clearly there is a pattern suggesting that the relationship should at least have been quadratic.

```
m2 <- lm(cty~trans+displ+I(displ^2),mpg1)
```

引入 displ 的平方作为模型中额外的预测变量

...results in a much less problematic set of residual plots, so we can interrogate the model further. For example the `summary` function can be used to report the parameter estimates, their standard deviations (standard errors), and the p-values associated with the hypothesis test that each true parameter value in turn is zero (the 't values' are the test statistic used for this).

```
> summary(m2)

Call:
lm(formula = cty ~ trans + displ + I(displ^2), data = mpg1)

Residuals:
    Min      1Q  Median      3Q     Max
-6.5988 -1.4620 -0.0174  1.1067 12.4922

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) 34.98583    1.27874  27.360   <2e-16 ***
transmanual  0.45431    0.32911   1.380    0.169
displ       -8.23552    0.71766 -11.475   <2e-16 ***
I(displ^2)   0.75213    0.09366   8.031    5e-14 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 2.257 on 230 degrees of freedom
Multiple R-squared:  0.7224,Adjusted R-squared:  0.7188
F-statistic: 199.5 on 3 and 230 DF,  p-value: < 2.2e-16
```

P值小，代表对模型的贡献大

Clearly there is strong evidence for the `displ`acement effect, but no real evidence of an additional effect of `trans`mission. Let's check whether that conclusion might result from being too simplistic about the `trans` effect, by allowing an interaction. And let's then test the null hypothesis that a model with no `trans` effect is correct, against the alternative that the full interaction model is needed, as follows:

```
> m3 <- lm(cty~(trans+displ+I(displ^2))^2,mpg1)
> m0 <- lm(cty~displ+I(displ^2),mpg1)
> anova(m0,m3) ## F ratio test of H_0: m0 is correct vs H_1: we need m3
Analysis of Variance Table

Model 1: cty ~ displ + I(displ^2)                    m0
Model 2: cty ~ (trans + displ + I(displ^2))^2        m3
  Res.Df    RSS Df Sum of Sq      F Pr(>F)
1    231 1181.1
2    227 1147.0  4    34.092 1.6867 0.1539
```

剩余平方和（越小，越好）

大于 0.05，说明模型 2 没有显著优于模型 1。

（如果 $H_0$ is true，我们有 15.39% 的概率看到 $F \geq 1.6867$，大于概率大于 5%，我们无法拒绝 $H_0$）

So no evidence for a `trans` effect of any sort. But what about this?

```
> summary(lm(cty~trans,mpg1))

Call:
lm(formula = cty ~ trans, data = mpg1)

Residuals:
    Min      1Q  Median      3Q     Max
-9.6753 -2.9682  0.0318  2.3247 16.3247

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)  15.9682     0.3248  49.168  < 2e-16 ***
transmanual   2.7072     0.5662   4.782 3.09e-06 ***
...
```

If we don't consider `displ` then there appears to be a strong and significant effect of `trans`. Why? Well, on examination of the data, it appears that manual cars tend to have smaller engines than automatic ones. And because smaller engines have lower fuel consumption, manual cars therefore appear to be more efficient. But if you control for engine size (`displ`) then there is no additional effect of transmission.

So are manual cars more efficient or not? That's not so easy to answer. For a given engine size it appears not, but for a given performance level the answer might well be yes: the automatics need a larger engine for the same performance, and therefore are less efficient for a given performance level. Clearly the interpretation of statistical model analyses require care. It requires particular care not to interpret effects as causal without alot more evidence than a statistical model of observational data can provide. In this example the `displ` variable that probably drives the apparent `trans` effect is one of the measured predictors. But in many cases it's quite possible that a variable we have not measured is driving both the response variable and the predictor, with no causal relationship between the response and the predictor at all.

This can really matter. For example there has been a good deal of work attempting to prove that lock down measures, and only lock down measures, are what controlled Covid-19 in various places at various times in the pandemic. But ultimately almost all that these studies show is an *association* between lockdown and reduced Covid transmission. It is impossible to rule out that the spontaneous changes in people's behaviour, which included clamouring for lockdowns on social media, were enough both to stop transmission increasing and to push governments into action. Nor to rule out that measures short of full lockdown, which were invariably associated with, but preceded, full lockdown, were what caused infections to start to decline.

截距项 表示 group 为参考组（ctrl）时的重量，P 值小于 2e-16，说明截距项显著。

F-statistic 为 4.846，P 值为 0.01591，小于 0.05，说明 group 对植物重量有显著影响。

**Exercise**

Use `lm` to fit the simple linear model for the `PlantGrowth` data in which `weight` depends on experimental treatment `group`. Check the residual plots for the model, and then test whether `weight` depends on `group`. If it does, interpret the model parameters. If the results are not clear cut, consider resetting the levels of the `group` factor to help.

```
data（PlantGrowth） # 加载数据
model ← lm（weight ~ group，data = PlantGrowth） # 拟合线性模型
summary（model） # 查看模型结果
plot（model）
```

# 10 R classes

The linear modelling routines provide a good introduction to *classes* in R. The `lm` function returns an object of class "`lm`", and typing the name of such an object produces the short summary of the model we saw above. What it doesn't do is to print the entire contents of the object. You can check that by looking at `str(m1)`, for example: `m1` is packed full of stuff. So what is happening when `m1` gets printed?

R provides a simple form of object orientation, where some functions are defined as *generic* functions, with *methods* specific to the class of their first argument. So the `print` function has a default method, but also several hundred other methods (type `methods(print)` to see them), including one for objects of class "`lm`", called `print.lm()`. It is this latter method that is invoked when you type `print(m1)`, or just `m1` at the R command prompt. Similarly `plot(m1)` actually uses the method function `plot.lm()`, and `summary(m1)` really calls

`summary.lm`. The function `residuals` is also generic, and again it is the method function `residuals.lm` that is invoked by `residuals(m1)`.

As a simple example, suppose we want to create a class, `"foo"`, of objects whose structure is a list, with elements `a` and `sd` representing some estimates and their standard deviations, and we want the print method for such objects to print a simple approximate confidence interval for the estimate and its standard deviation. Here is an appropriate print method function.

```
print.foo <- function(x) {
  cat(x$a-1.96*x$sd,"\n")
  cat(x$a+1.96*x$sd,"\n")
}
```

And here is an example of it in action

```
> a <- list(a = 1:3, sd = c(.1,.2,.1))
> class(a) <- "foo"
> print(a)
0.8 1.6 2.8
1.2 2.4 3.2
```

Note that an object can have several classes arranged in a hierarchy, usually when an object somehow extends or generalizes an existing class, and can therefore re-use some of the methods from that original class. This is referred to as *inheriting* from an existing class. For example R has a `glm` function for fitting *generalized linear models*. It returns objects of class `"glm"` *inheriting* from class `"lm"`. e.g.

```
> x <- runif(20); y <- rpois(20,exp(2*x))
> m <- glm(y~x,family=poisson)
> class(m)
[1] "glm" "lm"
```

When a method function is called with an argument of class `"glm"`, R first looks to see if there is a method function for class `"glm"` objects. If not then it looks for the method function for class `"lm"` objects, only resorting to the default method function if it finds neither of these. So, for example, `summary(m)` would invoke method function `summary.glm`, but `plot(m1)` invokes method function `plot.lm`, while `AIC(m1)` invokes method function `AIC.default`.

The fact that objects can inherit from several classes means that you should not test for class membership using code like `if (class(a)=="foo")` but rather use `if inherits(a,"foo")`.

## Exercise

Auto-differentiation (AD) is widely used in machine learning and statistics to simplify implementation of new models. The basic idea is to automatically compute the derivatives of a function based only on the code for evaluating the function. One way to do this is to use the ability to make operators and functions class specific. For example, suppose I want to evaluate $sin(x_1 x_2)$ and its derivatives w.r.t. $x_1$ and $x_2$. One way to do this would be to create a new class `"ad"` having a value and gradient attribute, the latter to contain the derivatives of the object w.r.t. some variables of interest, here $x_1$ and $x_2$. Then I can create version of the `sin` and $\star$ specific to this class that will automate application of the chain rule and known derivatives.

```
ad <- function(x,diff = c(1,1)) { ## create class "ad" object
  ## diff[1] is number of grads, diff[2] is element to set to 1.
  grad <- rep(0,diff[1])
  if (diff[2]>0 && diff[2]<=diff[1]) grad[diff[2]] <- 1
  attr(x,"grad") <- grad; class(x) <- "ad"; x
}
sin.ad <- function(a) {
  grad.a <- attr(a,"grad"); a <- as.numeric(a) ## avoid infinite recursion!
  d <- sin(a); attr(d,"grad") <- cos(a) * grad.a
  class(d) <- "ad"; d
}
```

```
exp.ad <- function(a) {
  grad.a <- attr(a,"grad")
  a <- as.numeric(a)
  d <- exp(a)
  attr(d,"grad") <- d * grad.a
  class(d) <- "ad"
  return(d)
}
```

```
"/.ad" <- function(a,b) {
  grad.a <- attr(a,"grad")
  grad.b <- attr(b,"grad")
  a <- as.numeric(a)
  b <- as.numeric(b)
  d <- a/b
  attr(d,"grad") <- (grad.a * b - grad.b * a)/(b^2)
  class(b) <- "ad"; return(b)
}
```

```
f.ad <- function(x1, x2, x3) {
  term1 <- x1 * x2 * sin(x3)
  term2 <- exp.ad(x1 * x2)
  result <- (term1 + term2) / x3
  return(result)
}
```

```
"*.ad" <- function(a,b) { ## ad multiplication
  grad.a <- attr(a,"grad"); grad.b <- attr(b,"grad")
  a <- as.numeric(a); b <- as.numeric(b); d <- a*b
  attr(d,"grad") <- a * grad.b + b * grad.a
  class(d) <- "ad"; d
}
## create ad objects with grad attr w.r.t. x1 and x2...
x1 <- ad(1,c(2,1)); x2 <- ad(2,c(2,2)); sin(x1*x2)
```

Try out the code, understand it, and then extend it to automatically differentiate the function $f(x_1, x_2, x_3) = \{x_1 x_2 \sin(x_3) + e^{x_1 x_2}\}/x_3$ w.r.t. its arguments.

# 11 Matrix computation

Both the linear model and the notion of tidy data rely on matrices, and in fact matrix computation is fundamental to many statistical modelling and statistical learning methods. You need to know some basics, starting with multiplication. You might think there is not much to say about this, but consider the following example:

```
> n <- 2000; y <- runif(n) ## example vector
> A <- matrix(runif(n*n),n,n) ## example matrix
> B <- matrix(runif(n*n),n,n) ## example matrix
> system.time(a1 <- A %*% B %*% y)
   user system elapsed
  1.435  0.003   1.629
> system.time(a2 <- A %*% (B %*% y))
   user system elapsed
  0.018  0.000   0.018
> range(a1-a2) ## results the same
[1] -2.852175e-09 3.026798e-09
```

Why is `A %*%B %*%y` so much slower than `A %*%(B %*%y)`? Consider the definition of matrix multiplication[14]:

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}.$$

Each element $C_{ij}$ involves $n$ multiplications and $n$ additions, and for the above case there are $n^2$ elements $C_{ij}$. So there are $2n^3$ computations to perform. Often we are not much concerned about constants such as 2, and simply write that the cost is $O(n^3)$ ('order $n^3$'). What about the matrix vector product?

$$a_i = \sum_{k=1}^{n} B_{ik} y_k,$$

so each of the $n$ elements of $\mathbf{a}$ also costs $n$ multiplications and $n$ additions. So the cost is now $O(n^2)$. The origin of the difference in timings should be clear. When forming `a1`, R worked from left to right, first forming the product of $\mathbf{A}$ and $\mathbf{B}$, and only then the product of the result with $\mathbf{y}$, so the dominant cost is $O(n^3)$. When forming `a2` the brackets force the vector $\mathbf{By}$ to be formed first, and then the product of $\mathbf{A}$ with the resulting matrix is formed. The cost of both steps is $O(n^2)$. This accounts for the speed up, but you will notice that the speed up was not by a factor of 2000. That's because of other overheads in the process, which are not affected by the reduction in the operations count.

In addition to matrix multiplication (`%*%`) there are several other basic operations:

`t()` transposes a matrix. So `t(A)` forms $\mathbf{A}^{\mathrm{T}}$.

`crossprod(A)` forms the crossproduct $\mathbf{A}^{\mathrm{T}}\mathbf{A}$ as efficiently as possible [15].

[14] you should have a mental picture of this, going across the rows of $\mathbf{A}$ and down the columns of $\mathbf{B}$.

[15] `t(A)%*%A` has about twice the computational cost. Why?

1. n ← 2000
A ← matrix(runif(n*n), n, n)
B ← matrix(runif(n*n), n, n)
tr_AB ← sum(A * t(B))
tr_AB

OR

for i in 1:n {
C ← A[i, ] * B[, i]
dia_AB[i] ← sum(C)
}
sum(dia_AB)

2. diag_AB ← rowSums(A * t(B))

`diag()` allows the leading diagonal of a matrix to be be extracted or set. e.g. `diag(A) <-1:3` sets the leading diagonal of $3 \times 3$ matrix $\mathbf{A}$ to the integers 1 to 3. `sdiag` in package `mgcv` does the same for sub- or super-diagonals.

Operators such as `+ - * /` operate element wise on matrices.

`drop(x)` drops dimensions of an array having only one level. e.g. if `x` is an $n \times 1$ matrix, `drop(x)` is an $n$ vector.

## Exercises

1. The *trace* of a matrix is the sum of its leading diagonal elements. i.e. $\mathrm{tr}(\mathbf{A}) = \sum_i A_{ii}$. Suppose you have two (compatible) matrices $\mathbf{A}$ and $\mathbf{B}$ and want to evaluated $\mathrm{tr}(\mathbf{AB})$. Clearly forming $\mathbf{AB}$ for this purpose is wasteful, as the off diagonal elements of the product contribute nothing. By considering the summation that defines $\mathrm{tr}(\mathbf{AB})$, find a way to evaluate it efficiently using regular multiplication, transposition and summation. Try it out with an R example. Give the order of the factor by which the efficiency is improved.

2. By making use of `rowSums` find an equally efficient way of computing $\mathtt{diag}(\mathbf{AB})$, and try it in R.

## 11.1 General solution of linear systems

Suppose we want to solve $\mathbf{Ab} = \mathbf{c}$ for $\mathbf{b}$, where $\mathbf{A}$ is an $n \times n$ full rank matrix, and $\mathbf{c}$ is an $n$ vector, or an $n \times m$ matrix ($\mathbf{b}$ is obviously of the same dimension).

`b <- solve(A,c)` b 是未知数

will do this. Formally the solution is $\mathbf{b} = \mathbf{A}^{-1}\mathbf{c}$, but we almost never form $\mathbf{A}^{-1}$ explicitly, as it is less efficient computationally, and the resulting $\mathbf{b}$ typically has higher numerical error. However, given a routine for solving linear systems, it is obviously possible to use it to solve $\mathbf{AA}^{-1} = \mathbf{I}$ for $\mathbf{A}^{-1}$, and `Ai <-solve(A)` will do just that. For most statistical work `solve` is suboptimal in various ways, so let's consider the matrix methods that do tend to be more useful.

## 11.2 Cholesky decomposition

A symmetric $n \times n$ matrix, $\mathbf{A}$, is *positive definite* if $\mathbf{x}^{\mathrm{T}}\mathbf{Ax} > 0$ for any non-zero $\mathbf{x}$. This is exactly equivalent to it having only positive eigenvalues. $\mathbf{A}$ is *positive semi-definite* if $\mathbf{x}^{\mathrm{T}}\mathbf{Ax} \geq 0$ for any non-zero $\mathbf{x}$ (equivalently all eigenvalues are $\geq 0$). Positive (semi) definite matrices are important in statistics because all covariance matrices are in this class. Recall that a covariance matrix, $\Sigma$, of a random vector $\mathbf{X}$ is the matrix containing the covariance of $X_i$ and $X_j$ as its element $\Sigma_{ij}$. That is $\Sigma_{ij} = E[\{X_i - E(X_i)\}\{X_j - E(X_j)\}]$, or equivalently

$$\mathbf{\Sigma} = E[\{\mathbf{X} - E(\mathbf{X})\}\{\mathbf{X} - E(\mathbf{X})\}^{\mathrm{T}}].$$

Various tasks are especially easy and efficient with positive definite matrices. This is largely because a positive definite matrix can be decomposed into the crossproduct of an upper triangular matrix with itself. That is we can write $\mathbf{\Sigma} = \mathbf{R}^{\mathrm{T}}\mathbf{R}$, where $\mathbf{R}$ is upper triangular, meaning that $R_{ij} = 0$ if $i > j$. It's easy to see how this works on a small example

$$\begin{bmatrix} R_{11} & 0 & 0 \\ R_{12} & R_{22} & 0 \\ R_{13} & R_{23} & R_{33} \end{bmatrix} \begin{bmatrix} R_{11} & R_{12} & R_{13} \\ 0 & R_{22} & R_{23} \\ 0 & 0 & R_{33} \end{bmatrix} = \begin{bmatrix} \Sigma_{11} & \Sigma_{12} & \Sigma_{13} \\ \Sigma_{12} & \Sigma_{22} & \Sigma_{23} \\ \Sigma_{13} & \Sigma_{23} & \Sigma_{33} \end{bmatrix}$$

Starting at top left, and working across the columns and down the rows we have

$$\begin{aligned}
\mathbf{R}_{11}^2 &= \Sigma_{11} \\
R_{11}\mathbf{R}_{12} &= \Sigma_{12} \\
R_{11}\mathbf{R}_{13} &= \Sigma_{13} \\
R_{12}^2 + \mathbf{R}_{22}^2 &= \Sigma_{22} \\
R_{12}R_{13} + R_{22}\mathbf{R}_{23} &= \Sigma_{23} \\
R_{13}^2 + R_{23}^2 + \mathbf{R}_{33}^2 &= \Sigma_{33}
\end{aligned}$$

where at each row the only unknown is shown in bold. For an $n \times n$ matrix this becomes (defining $\sum_{i=1}^{0} x_i \equiv 0$)

$$R_{ii} = \sqrt{\Sigma_{ii} - \sum_{k=1}^{i-1} R_{ki}^2} \text{ and } R_{ij} = \frac{\Sigma_{ij} - \sum_{k=1}^{i-1} R_{ki}R_{kj}}{R_{ii}}.$$

An immediate use is to compute the determinant

$$|\mathbf{\Sigma}| = |\mathbf{R}^{\mathrm{T}}\mathbf{R}| = |\mathbf{R}|^2 = \left(\prod_{i=1}^{n} R_{ii}\right)^2.$$

or better the log determinant $\log|\mathbf{\Sigma}| = 2\sum_{i=1}^{n} \log R_{ii}$.

Solving is also easy once we are dealing with triangular factors. To solve $\mathbf{\Sigma x} = \mathbf{y}$ for $\mathbf{x}$, we just need to solve $\mathbf{R}^{\mathrm{T}}\mathbf{R}\mathbf{x} = \mathbf{y}$. That is done by solving $\mathbf{R}^{\mathrm{T}}\mathbf{z} = \mathbf{y}$ for $\mathbf{z}$ and then $\mathbf{R}\mathbf{x} = \mathbf{z}$ for $\mathbf{x}$. Again a small example makes it clear how easy this is:

$$\begin{bmatrix} R_{11} & 0 & 0 \\ R_{12} & R_{22} & 0 \\ R_{13} & R_{23} & R_{33} \end{bmatrix} \begin{bmatrix} z_1 \\ z_2 \\ z_3 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}.$$

Obviously working from the top we have the following system, with one unknown (in bold) at each line

$$\begin{aligned}
R_{11}\mathbf{z}_1 &= y_1 \\
R_{12}z_1 + R_{22}\mathbf{z}_2 &= y_2 \\
R_{13}z_1 + R_{23}z_2 + R_{33}\mathbf{z}_3 &= y_3
\end{aligned}$$

forward solve

The generalization of this *forward solve* is obvious, as is the equivalent *back solve*, where we start from the last line and work back solving for $x_j \ldots$

$$\begin{bmatrix} R_{11} & R_{12} & R_{13} \\ 0 & R_{22} & R_{23} \\ 0 & 0 & R_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} z_1 \\ z_2 \\ z_3 \end{bmatrix}.$$

back solve

In R, function `chol` computes the Cholesky factor, while `backsolve` and `forwardsolve` solve with triangular factors. One advantage of the Cholesky factorization is speed. Solving linear systems of dimension $n$ costs $O(n^3)$ operations whether we use `chol` or `solve`, but the constant of proportionality is smaller for Cholesky. For example.

```
> n <- 2000
> A <- crossprod(matrix(runif(n*n),n,n)) ## example +ve def matrix
> b <- runif(n) ## example n vector
> system.time(c0 <- solve(A,b)) ## solve
   user system elapsed
  0.577 0.000 0.577
> system.time({R <- chol(A); ## solve with cholesky
+ c1 <- backsolve(R,forwardsolve(t(R),b))})
   user system elapsed
  0.312 0.000 0.313
> range(c0-c1)/norm(A) ## confirm same result
[1] -3.673905e-12 3.970016e-12
```

$R^T R c = b$

$Rc = z$

$R^T z = b$

forward solve 求解 z

backsolve 求解 c

As an example, consider evaluating the log of the multivariate normal p.d.f.

$$-\frac{1}{2}(\mathbf{y}-\boldsymbol{\mu})^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}(\mathbf{y}-\boldsymbol{\mu})-\frac{1}{2}\log|\boldsymbol{\Sigma}|-\frac{n}{2}\log(2\pi)$$

Doing this by using R functions `solve` and `determinant` (or `det`) is much slower than is necessary. In particular `determinant` requires $O(n^3)$ operations, whereas once we have the Cholesky factor, determinant calculation is $O(n)$. Notice also how

$$(\mathbf{y}-\boldsymbol{\mu})^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}(\mathbf{y}-\boldsymbol{\mu})=\mathbf{z}^{\mathrm{T}}\mathbf{z} \text{ where } \mathbf{z}=\mathbf{R}^{-\mathrm{T}}(\mathbf{y}-\boldsymbol{\mu})$$

This suggests the following function (recall $|\mathbf{AB}|=|\mathbf{A}||\mathbf{B}|$, so $|\boldsymbol{\Sigma}|=|\mathbf{R}|^2$)

```
ldmvn <- function(y,m,S) {
## Cholesky based evaluation of log density of MVN at y.
## m is mean and S the covariance matrix.
  R <- chol(S) ## Get Cholesky factorization R'R = S
  z <- forwardsolve(t(R),y-m) ## R^{-T}(y-m)
  -sum(z^2)/2 - sum(log(diag(R))) - length(m)*log(2*pi)/2
} ## ldmvn
```

Here is a small example using it

```
> y <- c(3,4.5,2); m <- c(.6,.8,1.2)
> S <- matrix(c(1,.5,.3,.5,2,1,.3,1,1.5),3,3)
> ldmvn(y,m,S)
[1] -8.346052
```

## Exercises

1. If $\mathbf{y}=\mathbf{A}\mathbf{x}$ and $\mathbf{x}$ has covariance matrix $\boldsymbol{\Sigma}_x$, show that the covariance matrix of $\mathbf{y}$ is $\boldsymbol{\Sigma}_y=\mathbf{A}\boldsymbol{\Sigma}_x\mathbf{A}^{\mathrm{T}}$. This is a basic and fundamental result re-used repeatedly in statistics and data science.
2. A linear transformation of multivariate normal random variables is also multivariate normal. Write a function to generate `n` draws from a multivariate normal distribution with mean `m` and covariance matrix `S`, by transformation of independent $N(0,1)$ deviates, generated by `rnorm`. Also write code to test that it is working (function `cov` might be helpful).

## 11.3 QR decomposition

An orthogonal matrix $\mathbf{Q}$ is one with the property that $\mathbf{Q}^{\mathrm{T}}\mathbf{Q}=\mathbf{Q}\mathbf{Q}^{\mathrm{T}}=\mathbf{I}$. Any full rank square matrix, $\mathbf{A}$, can be decomposed into the product of an orthogonal matrix and an upper triangular matrix: $\mathbf{A}=\mathbf{Q}\mathbf{R}$. So the solution to $\mathbf{A}\boldsymbol{\beta}=\mathbf{y}$ is the solution to $\mathbf{R}\boldsymbol{\beta}=\mathbf{Q}^{\mathrm{T}}\mathbf{y}$ — it involves multiplication by an orthogonal matrix, and a back-solve.

But suppose that instead of full rank square matrix $\mathbf{A}$ defining the l.h.s. of the system we want to solve, we have $n\times p$, rank $p$, matrix $\mathbf{X}$, where $n>p$. Clearly we can not expect to find a solution to $\mathbf{X}\boldsymbol{\beta}=\mathbf{y}$ in general, since we have more equations to satisfy than there are elements of $\boldsymbol{\beta}$. But what about $\mathbf{X}\boldsymbol{\beta}\approx\mathbf{y}$? We can solve that provided we are a little more precise about what '$\approx$' means. In particular consider finding $\boldsymbol{\beta}$ to minimize the mean square difference between $\mathbf{X}\boldsymbol{\beta}$ and $\mathbf{y}$. That is we want to minimize the squared Euclidean length of $\mathbf{y}-\mathbf{X}\boldsymbol{\beta}$, the sum of squares of differences between $\mathbf{y}$ and $\mathbf{X}\boldsymbol{\beta}$. i.e. we seek

$$\hat{\boldsymbol{\beta}}=\underset{\beta}{\operatorname{argmin}}\|\mathbf{y}-\mathbf{X}\boldsymbol{\beta}\|^2$$

where $\|\mathbf{v}\|^2=\mathbf{v}^{\mathrm{T}}\mathbf{v}=\sum_{i=1}^n v_i^2$ is the squared Euclidean norm/Euclidean length of a vector, $\mathbf{v}$.

Solution of this problem involves the fact that we can also decompose $\mathbf{X}$ into the product of an $n\times n$ orthogonal matrix, $\boldsymbol{\mathcal{Q}}$, and an upper triangular matrix, so that

$$\mathbf{X}=\boldsymbol{\mathcal{Q}}\begin{bmatrix}\mathbf{R}\\ \mathbf{0}\end{bmatrix}$$

A n×n QR = A O(n³) [QᵀQ = I = QQᵀ]
orthogonal upper tri

$\mathbf{R}$ is $p \times p$ upper triangular. Defining $\mathbf{Q}$ to be the first $p$ columns of $\mathcal{Q}$, we can also write the decomposition as $\mathbf{X} = \mathbf{QR}$, of course. Now a defining property of orthogonal matrices is that they act as rotations/reflections, that is $\|\mathcal{Q}\mathbf{v}\|^2 = \|\mathbf{v}\|^2$, for any $\mathbf{v}$. It follows that

这个残差来自于 Q^T y 的后 (n-p) 行，与 beta 无关，所以可以忽略

$$\|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\|^2 = \|\mathcal{Q}^{\mathrm{T}}\mathbf{y} - \mathcal{Q}^{\mathrm{T}}\mathbf{X}\boldsymbol{\beta}\|^2 = \left\|\mathcal{Q}^{\mathrm{T}}\mathbf{y} - \begin{bmatrix}\mathbf{R}\\ \mathbf{0}\end{bmatrix}\boldsymbol{\beta}\right\|^2 = \|\mathbf{Q}^{\mathrm{T}}\mathbf{y} - \mathbf{R}\boldsymbol{\beta}\|^2 + \|\mathbf{r}\|^2$$

where $\mathbf{r}$ is the vector containing the final $n - p$ elements of $\mathcal{Q}^{\mathrm{T}}\mathbf{y}$. Since $\mathbf{r}$ does not depend on $\boldsymbol{\beta}$, then $\hat{\boldsymbol{\beta}}$ must be the minimizer of $\|\mathbf{Q}^{\mathrm{T}}\mathbf{y} - \mathbf{R}\boldsymbol{\beta}\|^2$. But this can be reduced to zero by setting

$$\mathbf{R}\hat{\boldsymbol{\beta}} = \mathbf{Q}^{\mathrm{T}}\mathbf{y} \quad \Rightarrow \quad \hat{\boldsymbol{\beta}} = \mathbf{R}^{-1}\mathbf{Q}^{\mathrm{T}}\mathbf{y}.$$

$\hat{\boldsymbol{\beta}}$ is the least squares estimate of the parameters of a linear model, and the given formula is how least squares estimates are computed in practice (obviously using a triangular solve, not explicit formation of $\mathbf{R}^{-1}$). This approach has much better numerical stability than using the theoretical formula $\hat{\boldsymbol{\beta}} = (\mathbf{X}^{\mathrm{T}}\mathbf{X})^{-1}\mathbf{X}^{\mathrm{T}}\mathbf{y}$ that you may have seen previously. It is easy to show that $\hat{\boldsymbol{\beta}}$ is unbiased (that is $E(\hat{\boldsymbol{\beta}}) = \boldsymbol{\beta}$). Further, since the covariance matrix of $\mathbf{y}$ is $\mathbf{I}\sigma^2$, the basic result on the covariance matrix of a linear transform shows that the covariance matrix of $\hat{\boldsymbol{\beta}}$ is

ε ~ N(0, Iσ²)

$$\boldsymbol{\Sigma}_{\hat{\boldsymbol{\beta}}} = \mathbf{R}^{-1}\mathbf{R}^{-\mathrm{T}}\sigma^2.$$

An unbiased estimate of $\sigma^2$ is $\hat{\sigma}^2 = \|\hat{\boldsymbol{\epsilon}}\|^2/(n-p) = \|\mathbf{y} - \mathbf{X}\hat{\boldsymbol{\beta}}\|^2/(n-p) = \|\mathbf{r}\|^2/(n-p)$.

The R function `qr` computes a QR decomposition. It returns an object of class `"qr"` containing the decomposition stored in a compact form. The components of the decomposition can be accessed using functions `qr.R` and `qr.Q` where the latter has optional argument `complete` which is set to `TRUE` if the full $n \times n$ matrix $\mathcal{Q}$ is required, rather than the default $n \times p$ matrix $\mathbf{Q}$. In fact explicitly extracting $\mathcal{Q}$ or $\mathbf{Q}$ is computationally very inefficient, if all we need is the product of a vector and the orthogonal factor. In that case it is far more efficient to use functions `qr.qy` and `qr.qty` which take the `qr` object as first argument and a vector, `y`, as second argument, and return respectively $\mathcal{Q}\mathbf{y}$ or $\mathcal{Q}^{\mathrm{T}}\mathbf{y}$.

Here are the computations for obtaining least squares estimates, applied to the simple `PlantGrowth` model from section 9. It uses the fact that $\mathbf{Q}^{\mathrm{T}}\mathbf{y}$ is the first $p$ elements of $\mathcal{Q}^{\mathrm{T}}\mathbf{y}$

```
> X <- model.matrix(~group,PlantGrowth) ## model matrix (note: response not needed!)
> qrx <- qr(X)          ## get QR decomposition of model matrix
> y <- PlantGrowth$weight ## response
> p <- ncol(X)          ## number of parameters
> beta <- backsolve(qr.R(qrx),qr.qty(qrx,y)[1:p]) ## R^{-1} Q^T y
> beta                  ## least squares estimates
[1] 5.032 -0.371 0.494
```

qr(X) 并不直接返回完整的 Q 和 R 矩阵，而是返回一个包含若干组件的列表，用于更高效地存储和计算这些矩阵。

This is exactly the computational approach used by `lm`.

1.
n <- 5
R <- qr.R(qr(matrix(runif(n * n), n, n))) ## get an R
Ri <- backsolve(R, diag(ncol(R))) ## solve R %*% Ri = I
range(R %*% Ri - diag(n)) ## check R %*% Ri = I

chol2inv(R) # 用来求矩阵A的逆矩阵
Ri %*% t(Ri)

## Exercises

1. Occasionally explicit inverses are needed. Write code that uses `backsolve` to find an explicit inverse of an upper triangular matrix, $\mathbf{R}$, and test it. Then look at `?chol2inv` and compare its results to your code.

2. Using the fact that $|\mathbf{AB}| = |\mathbf{A}||\mathbf{B}|$ and $|\mathbf{A}| = |\mathbf{A}^{\mathrm{T}}|$, show that for an orthogonal matrix, $\mathbf{Q}$, $|\mathbf{Q}| = \pm 1$. Hence run the code `set.seed(0);n<-100; A <-matrix(runif(n*n),n,n)` and use the QR decomposition to evaluate the magnitude[16] of the determinant of `A`. Compare to `determinant(A,log=FALSE)`. Now increase the size of the elements of `A` using `A <-A*1000` and repeat the exercise. This should emphasis the problem with working directly with the determinant, so now use the QR decomposition of `A` to obtain the log of the magnitude of the determinant of `A`.

标准化的Q，|Q|=1 而且这里我们只关心量级，不关心正负

[16] The sign of the determinant depends on whether $|\mathbf{Q}|$ is positive or negative. This can also be determined cheaply, but takes us too far afield: basically $\mathbf{Q}$ is constructed from the product of $n-1$, rank one orthogonal *Householder* matrices, each with determinant -1, so $|\mathbf{Q}| = (-1)^{n-1}$.

2.
set.seed(0)
n <- 100
A <- matrix(runif(n * n), n, n)
R <- qr.R(qr(A))
prod(abs(diag(R)))
determinant(A, log = FALSE)

A <- A * 1000
R <- qr.R(qr(A))
prod(abs(diag(R))) ## overflow
determinant(A, log = FALSE) ## overflow

sum(log(abs(diag(R)))) ## fine
determinant(A) ## fine

## 11.4 Pivoting and triangular factorizations

When computing factorizations involving triangular matrices, it turns out that the numerical stability of the factorization is affected by the order in which the rows and/or columns appear in the original matrix. But for all least squares and equation solving tasks the order in which rows and columns are present in the matrix doesn't actually matter - we can always re-order and get the same result (provided we take account of the re-ordering). So rows and/or columns can be re-ordered for maximum numerical stability - that is for minimum numerical error in the final answer. This is known as *pivoting*. `chol` and `qr` have options allowing pivoting to be used. Using them requires extra book-keeping, and takes us too far off topic in this course: but for serious computation you should be aware that pivoting is possible, and can be necessary.

## 11.5 Symmetric eigen-decomposition

Any real symmetric matrix, $\mathbf{A}$, has an *eigen-decomposition*

$$\mathbf{A} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^{\mathrm{T}}$$

where $\mathbf{U}$ is an orthogonal matrix, the columns of which are the (normalized) eigenvectors of $\mathbf{A}$, and $\mathbf{\Lambda}$ is a diagonal matrix of the eigenvalues of $\mathbf{A}$, arranged so that $\Lambda_{ii} \geq \Lambda_{i+1,i+1}$. R function `eigen` computes eigen-decompositions, as we saw in section 5.6. Rather than provide further simple examples of the use of `eigen`, let's look at a major application in multivariate statistics: principle components analysis.

### 11.5.1 Eigen-decomposition of a covariance matrix: PCA

The eigen-decomposition of a covariance matrix is often known as Principle Components Analysis (PCA). Consider a set of observations, $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$ of a $p$ dimensional random vector $\boldsymbol{\mathcal{X}}$, and suppose we create an $n \times p$ matrix $\mathbf{X}$ whose ith row is $\mathbf{x}_i$. The estimated variance of $\mathcal{X}_j$ is $\sum_{i=1}^{n}(X_{ij} - \bar{X}_j)^2/(n-1)$, and the estimated covariance of $\mathcal{X}_j$ and $\mathcal{X}_k$ is $\sum_{i=1}^{n}(X_{ij} - \bar{X}_j)(X_{ik} - \bar{X}_k)/(n-1)$ where $\bar{X}_j = n^{-1}\sum_{i=1}^{n} X_{ij}$. Hence if $\tilde{\mathbf{X}}$ is $\mathbf{X}$ with the column mean subtracted from each column, then the estimated sample covariance matrix for $\boldsymbol{\mathcal{X}}$ is simply

$$\hat{\mathbf{\Sigma}}_{\mathcal{X}} = \tilde{\mathbf{X}}^{\mathrm{T}}\tilde{\mathbf{X}}/(n-1).$$

每一列通常代表一个变量（**特征**），每一行代表一个样本

We can form an eigen decomposition $\hat{\mathbf{\Sigma}}_{\mathcal{X}} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^{\mathrm{T}}$ ($\mathbf{\Lambda} = \mathrm{diag}(\boldsymbol{\lambda})$ and $\lambda_i > \lambda_{i+1}$), and use this to define a new random vector $\mathbf{z} = \mathbf{U}^{\mathrm{T}}\boldsymbol{\mathcal{X}}$. Now $\mathbf{U}$ is orthogonal, so using the standard result for a linear transformation of a covariance matrix, we have that the estimated covariance matrix for $\mathbf{z}$ is

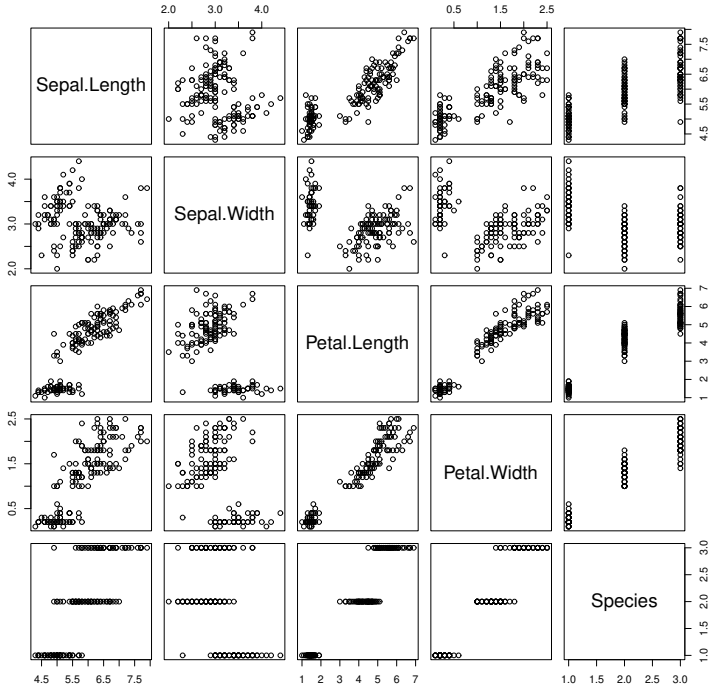
$$\hat{\mathbf{\Sigma}}_z = \mathbf{U}^{\mathrm{T}}\hat{\mathbf{\Sigma}}_{\mathcal{X}}\mathbf{U} = \mathbf{U}^{\mathrm{T}}\mathbf{U}\mathbf{\Lambda}\mathbf{U}^{\mathrm{T}}\mathbf{U} = \mathbf{\Lambda}.$$

i.e. the elements of $\mathbf{z}$ are uncorrelated, and the estimated variance of $z_j$ is $\lambda_j$. By the orthogonality of $\mathbf{U}$ we can write $\mathbf{U}\mathbf{z} = \boldsymbol{\mathcal{X}}$. i.e. $\boldsymbol{\mathcal{X}}$ is a weighted sum of the orthogonal columns of $\mathbf{U}$, where the weights are uncorrelated random variables with variances given by the $\lambda_j$. Hence the variability of $\boldsymbol{\mathcal{X}}$ has been decomposed additively into orthogonal *principle components*, $\mathbf{U}_{\cdot j}$ with associated variance $\lambda_j$. The principle component with the largest variance, $\lambda_1$, contributes the most variance, the component with the next largest variance $\lambda_2$ is next, and so on.

We can of course view $z_j$ as the co-ordinate of $\boldsymbol{\mathcal{X}}$ on the axis defined by $\mathbf{U}_{\cdot j}$, which suggests computing such co-ordinates for each original observations. That is computing $\mathbf{Z} = \mathbf{X}\mathbf{U}$ where the ith row of $\mathbf{Z}$ gives the co-ordinates of observation $\mathbf{x}_i$ relative to the axes defined by $\mathbf{U}$. Now, by construction these co-ordinates have sample variance $\lambda_1, \lambda_2, \ldots, \lambda_p$, with the first co-ordinate having most variability, the next having the second most variability and so on. This implies that we can use the co-ordinates for *dimension reduction*. Retaining the first $m$ components will capture $\sum_{i=1}^{m}\lambda_i / \sum_{i=1}^{p}\lambda_i$ of the variance in the original data. Often this proportion can be very high for $m \ll p$. This dimension reduction is equivalent to replacing the covariance matrix $\hat{\mathbf{\Sigma}}_{\mathcal{X}}$ with $\mathbf{U}_m\mathbf{\Lambda}_m\mathbf{U}_m^{\mathrm{T}}$ where $\mathbf{U}_m$ denotes the first $m$ columns of $\mathbf{U}$ and $\mathbf{\Lambda}_m$ the first $m$ rows and columns of $\mathbf{\Lambda}$.

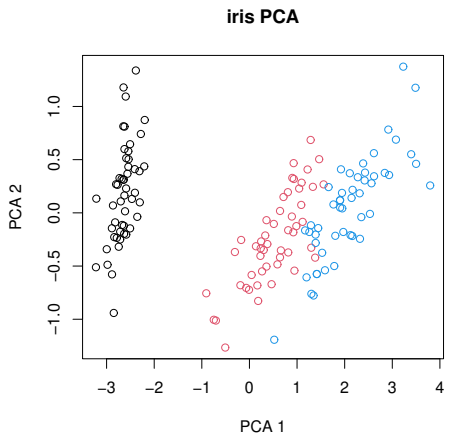
The classic dataset for illustrating this is the `iris` data, available in R. This gives 4 measurements for each of 50 iris flowers of 3 species (so $n = 150$, $p = 4$). Plotting the raw data is is not clear to what extent the measurements can really separate the species:

```
pairs(iris)
```




Let's see if the principle components do any better. Function `sweep` can be used to column centre the data matrix.

```
X <- sweep(as.matrix(iris[,1:4]),2,colMeans(iris[,1:4])) ## col centred data matrix
ec <- eigen(t(X)%*%X/(nrow(X)-1)) ## eigen decompose the covariance matrix
U <- ec$vectors;lambda <- ec$values ## exract eigenvectors and values
Z <- X %*% U;                ## the principle co-ordinated
plot(Z[,1],Z[,2],col=c(1,2,4)[as.numeric(iris$Species)],main="iris PCA",
    xlab="PCA 1",ylab="PCA 2") ## plot first two components
```




The three species are quite clearly separated now. Notice how the range of the first component is substantially greater than that of the second component — as expected, since the components are in decreasing order of variance explained. Here is some code to re-iterate the link between the $\mathbf{Z}$ column variances and $\boldsymbol{\lambda}$, and assess what proportion of the variance is explained by (can be attributed to) the first 2 principle components (columns of $\mathbf{Z}$).

```
X <- sweep(as.matrix(iris[, 1:4]), 2, colMeans(iris[, 1:4])) ## column centred data matrix
V <- t(X) %*% X / (nrow(X) - 1) ## estimated covariance matrix
sd <- diag(V)^.5 ## standard deviation
C <- t(t(V / sd) / sd) ## form C = diag(1/sd) %*% V %*% diag(1/sd) efficiently
## Actually cor(iris[,1:4]) does the same!!
ec <- eigen(C) ## eigen decompose the correlation matrix
U <- ec$vectors
lambda <- ec$values ## exract eigenvectors and values
Z <- X %*% U
## the principle co-ordinated
plot(Z[, 1], Z[, 2],
    col = c(1, 2, 4)[as.numeric(iris$Species)], main = "iris PCA",
    xlab = "PCA 1", ylab = "PCA 2"
) ## plot first two components
sum(lambda[1:2]) / sum(lambda)
## ... first 2 components explain lower proportion than with cov based PCA
```

```
> apply(Z,2,var);lambda ## compare Z column variances to lambda
[1] 4.22824171 0.24267075 0.07820950 0.02383509
[1] 4.22824171 0.24267075 0.07820950 0.02383509
> sum(lambda[1:2])/sum(lambda) ## proportion variance explained by components 1 and 2
[1] 0.9776852
```

### Exercise

An obvious alternative is to standardize the original variables to have unit variance and then perform PCA. In that case we are really working with the correlation matrix for the variables, rather than the covariance matrix. Try this and compare the separation of the species now. Examining the eigenvalues, suggest a possible partial explanation for any difference.

## 11.6 Singular value decomposition

PCA is an example of *dimension reduction*: finding a lower dimensional representation of high dimensional data. In a sense much of statistics and data-science is about dimension reduction: about finding ways of representing complex data using simpler summaries, without the loss of important information. But for the moment, let's stick with finding lower dimensional representations of matrices. One output of PCA was low rank approximations to covariance matrices - but what about low rank approximations to non-square matrices? *Singular value decomposition* gives a way to achieve this. Any $n \times p$ matrix, $\mathbf{X}$ can be decomposed

$$\mathbf{X} = \mathbf{U}\mathbf{D}\mathbf{V}^{\mathrm{T}}$$

where $\mathbf{U}$ is an $n \times p$ matrix with orthogonal columns, $\mathbf{V}$ is an orthogonal matrix, and $\mathbf{D}$ is a diagonal matrix of the *singular values* of $\mathbf{X}$ — the positive square roots of the eigenvalues of $\mathbf{X}^{\mathrm{T}}\mathbf{X}$. $\mathbf{D}$ is arranged so that $D_{ii} \geq D_{i+1,i+1}$ (that is the singular values are in decreasing order on the diagonal). Suppose $\mathbf{U}_r$ and $\mathbf{V}_r$ denote the matrices consisting of the first $r$ columns of $\mathbf{U}$ and $\mathbf{V}$, while $\mathbf{D}_r$ is the first $r$ rows and columns of $\mathbf{D}$. Then

$$\mathbf{U}_r\mathbf{D}_r\mathbf{V}_r^{\mathrm{T}}$$

is the best rank $r$ approximation to $\mathbf{X}$ (in a sense that is beyond our scope here). The `svd` function in R returns the singular value decomposition of a matrix.

### Exercise

(X^TX)^{-1}X^Ty = (VD^2V^T)^{-1}VDU^Ty=VD^{-2}V^TVDU^Ty = VD^{-1}U^Ty

The formal expression for the least squares estimates of a linear model is $\hat{\boldsymbol{\beta}} = (\mathbf{X}^{\mathrm{T}}\mathbf{X})^{-1}\mathbf{X}^{\mathrm{T}}\mathbf{y}$. Find a simplified expression for this in terms of the components of the SVD. Use you expression to compute the least squares estimates for the `PlantGrowth` model of section 11.3, one more time.

```
X <- model.matrix(~group, PlantGrowth)
sx <- svd(X)
beta.hat <- sx$v %*% ((t(sx$u) %*% PlantGrowth$weight) / sx$d)
beta.hat
coef(lm(weight ~ group, PlantGrowth))
```

# 12 Design, Debug, Test and Profile

Having covered enough R, and a sufficient diversity of statistically oriented examples, that writing code for serious statistical tasks is possible, it is time to consider some topics that are a essential for larger coding projects.

## 12.1 Design

As mentioned several times already in these notes, you will write better code more quickly if you spend some time and effort thinking about, planning and writing down the *design* of your code, before you type any actual code at the keyboard. A major aim of code design is to achieve well *structured* code, in which the overall computational task has been broken down into logical components, coded in concise manageable functions, each with well defined *testable* properties. Well structured programmes are are easier to code, test, debug, read and maintain.

For complex projects, design may be multi-levelled, with the task broken down into 'high level' logical units, with each unit requiring further design work itself. But however complex the task, the same basic approach is useful at each level.

1. *Purpose*. Write down what your code should do. What are the inputs and outputs, and how do you get from one to the other?

2. *Design Considerations*. What are the other important issues to bear in mind? e.g. does your code need to be quick? or deal with large data sets? is it for one off use, by you, or for general use by others? etc.

3. *Design and structure*. Decide how best to split the code into functions each completing a well defined manageable and testable part of the overall task. Write down the structure.

4. *Review*. Will this structure achieve the purpose and meet the design considerations?

Often you will need to iterate and refine the process a few times to achieve a design you are happy will do the job, and be as straightforward as possible to code. And some parts of any design are likely to need more careful specification and detailing than others.

At the design stage you are acting like the architect and structural engineer for a building. Once you start to code you are acting as the builder. This is a useful analogy. Architects' plans never cover how every brick is to be laid. The builder still has to solve many on the ground problems during construction, but the better the plans, the more smoothly the construction phase tends to go. Just as making up any but the simplest building work as you go along is a recipe for making a mess, so is trying to design as you code - at least sketch out a plan first. But also expect to have to modify your plan as you code, as some realities become apparent that were not anticipated.

Good planning and structure also help to write readable code, but readability also requires good commenting. If you write reasonable good code then as you write it, everything about your code will usually seem very obvious to you. But two weeks later it will not seem obvious. Neither will it seem obvious to anyone else (as the team-working experience from this course so far will hopefully have already made clear). You should therefore aim to comment your code so that you could give it to another R programmer and they could quickly understand what is supposed to be doing and how it does it, *without knowing anything about these things previously*. You can help others and your later self by

1. Using (short) meaningful variable names where possible.

2. Explaining what the code does using frequent `# comments`, including a short overview at the start of each function, and in longer projects an overview of the code's overall purpose.

3. Laying out the code so that it is clear to read. Remember that white space and blank lines cost nothing.

4. Whenever you work in a team, always ensure that any piece of code and commenting written by one team member is carefully reviewed and checked by another.

## 12.2 Ridge regression — a very short design example

The parameters estimates, $\hat{\boldsymbol{\beta}}$, for the linear model $\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}$ can become rather unstable if $p$, the number of parameters, becomes large relative to $n$, the number of data. Indeed if $p > n$ then least squares parameter estimates $\hat{\boldsymbol{\beta}}$, will not exist. This in turn means that model predictions $\hat{\mu} = \mathbf{X}\hat{\boldsymbol{\beta}}$ may poorly approximate the true $\boldsymbol{\mu} = E(\mathbf{y})$. One approach that can help is *ridge regression*, which reduced variability in $\hat{\boldsymbol{\beta}}$ by *shrinking* parameters towards zero. Specifically we estimate parameters as

懲罰項

$$\hat{\boldsymbol{\beta}} = \underset{\boldsymbol{\beta}}{\operatorname{argmin}} \|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\|^2 + \lambda\|\boldsymbol{\beta}\|^2 = (\mathbf{X}^{\mathrm{T}}\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^{\mathrm{T}}\mathbf{y}$$

for some *smoothing parameter* $\lambda$. The larger we make $\lambda$ the more the elements of $\hat{\boldsymbol{\beta}}$ are shrunk towards zero, while $\lambda = 0$ returns the least squares estimates[17]. $\lambda$ can be selected to optimize an estimate of the error that the model would make in predicting a replicate dataset to which it had not been fitted. One such estimate is the *generalized cross validation* score,

$$V(\lambda) = \frac{n\|\mathbf{y} - \mathbf{X}\hat{\boldsymbol{\beta}}\|^2}{\{n - \text{trace}(\mathbf{A})\}^2},$$

[17] There is a close link between ridge regression and a Bayesian approach to linear modelling, in which a prior $\beta \sim N(\mathbf{0}, \mathbf{I}\lambda^{-1}\sigma^2)$ is placed on the linear model parameters. With such a prior, $\hat{\boldsymbol{\beta}}$, is the posterior mode for $\boldsymbol{\beta}$.

```
fit.ridge <- function(y, X, XX, sp) {
  ## fits model y = X b + e by ridge regression, with penalty parameter sp
  ## and computes GCV score.
  p <- ncol(X)
  n <- nrow(X)
  ## Compute hat matrix... tr(BD) = tr(DB)
  ## tr{X(X'X+sI)^{-1}X'} = tr{(X'X+sI)^{-1}X'X}
  A1 <- solve(XX + sp * diag(p), XX)
  trA <- sum(diag(A1)) ## Effective degrees of freedom
  ## compute coeff estimates...
  b.hat <- solve(XX + sp * diag(p), drop(t(y) %*% X))
  fv <- X %*% b.hat ## fitted values
  gcv <- sum(n * (y - fv)^2) / (n - trA)^2
  list(b.hat = b.hat, fv = fv, gcv = gcv, edf = trA)
} ## fit.ridge
```

where $\mathbf{A} = \mathbf{X}(\mathbf{X}^{\mathrm{T}}\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^{\mathrm{T}}$, so that $\hat{\boldsymbol{\mu}} = \mathbf{A}\mathbf{y}$. The term trace$(\mathbf{A})$ is the *effective* degrees of freedom of the model. It varies smoothly between $p$ and 0 as $\lambda$ is varied between 0 and $\infty$.

Let's briefly consider the design a simple ridge regression function.

1. *Aim*. Produce a function that takes model matrix, $\mathbf{X}$, response vector $\mathbf{y}$, a sequence of $\lambda$ values to try, and returns the vector of ridge regression parameter estimates $\hat{\boldsymbol{\beta}}$ corresponding to the GCV optimal choice of $\lambda$.

2. *Considerations*. Reasonable to assume that inputs are correct here. Important to check that the GCV score has a proper minimum. Since ridge regression often used for large $p$, efficiency is of some importance.

3. *Outline high level design*:

   (a) Top level function `ridge(y,X,lsp)` loops through log smoothing parameters in `lsp`, calling function `fit.ridge` for each to evaluate GCV score. Hence find optimal log smoothing parameter. Plots GCV against `lsp` using function `plot.gcv`. Returns optimal $\hat{\boldsymbol{\beta}}$

   (b) `fit.ridge(y,X,sp)` fits ridge regression as above using Cholesky[18] based method to solve for $\hat{\boldsymbol{\beta}}$. `sp` is the $\lambda$ value to use. Return the GCV score, effective degrees of freedom and $\hat{\boldsymbol{\beta}}$.

   (c) `plot.gcv(edf,lsp,edf)` plots the GCV score against effective degrees of freedom, `edf` and log smoothing parameters, `lsp`, marking the minimum.

So the overall task has been broken down into simpler tasks for which testable functions can be written.

```
plot.gcv <- function(edf, lsp, gcv) {
  ## plot GCV against EDF and log s.p. showing minimum
  par(mfrow = c(1, 2), mar = c(4, 4, 1, 1))
  plot(lsp, gcv, xlab = expression(log(lambda)), ylab = "GCV",
type = "l")
  i.min <- which(gcv == min(gcv))
  points(lsp[i.min], gcv[i.min], col = 2, pch = 19)
  plot(edf, gcv, xlab = "EDF", ylab = "GCV", type = "l")
  points(edf[i.min], gcv[i.min], col = 2, pch = 19)
}
```

### Exercise

Complete any necessary design details for each function, and implement `ridge`.

## 12.3 Testing

It is important to test code. You should of course test code as you write it, checking that it does what you expect in an interactive manner, but you should also devote some time to developing tests that can be run repeatedly in a more automated manner. One important approach is known as *unit testing*, where you write code implementing tests that check whether the components of a software project do what they are designed to do. In R programming this would usually mean writing code to check that the functions that make up your project each perform as expected, when supplied with arguments covering the range of conditions that the function is designed to cope with (and often also testing that appropriate error message are given when inappropriate inputs are provided).

In complex projects it is also a good idea to design appropriate tests that the functions are working together in the appropriate way, although this is often somewhat automatic, since projects are often hierarchical, with some functions calling several other functions, so that tests of the calling function must automatically test the co-ordinated operation of the functions being called.

As a very simple example, consider unit tests for the simple matrix argument to matrix value function from section 5.6, modified slightly to also include some error checking.

```
mat.fun <- function(A,fun=I) {
  if (!is.matrix(A)) A <- matrix(A) ## coerce to matrix if not already a matrix
  if (ncol(A)!=nrow(A)) stop("matrix not square") ## halt execution in this case
  if(max(abs(A-t(A)))>norm(A)*1e-14) stop("matrix not symmetric")
  ea <- eigen(A,symmetric=TRUE) ## get eigen decomposition
  ## generalize 'fun' to matrix case...
  ea$vectors %*% (fun(ea$values)*t(ea$vectors)) ## note use of re-cycling rule!
}
```

We might write tests for the case where `fun` is the identity, square root and inverse, where in each case the returned value should be easily checked. Another test might check that non-square matrices are correctly handled.

[18] Actually QR based methods would be more stable, but involve extra maths that is unhelpful for this illustration.

```
ridge <- function(y, X, lsp = seq(-5, 5, length = 100)) {
  ## function for ridge regression of y on X with generalized
  ## cross validation lsp is the set of smoothing parameters to try
  edf <- gcv <- lsp * 0 ## vectors for edf and gcv
  XX <- crossprod(X)
  for (i in 1:length(lsp)) { ## loop over log smoothing parameters
    fr <- fit.ridge(y, X, XX, exp(lsp[i])) ## fit
    gcv[i] <- fr$gcv
    edf[i] <- fr$edf
  }
  plot.gcv(edf, lsp, gcv) ## plot results
  i.opt <- max(which(gcv == min(gcv)))
  ## locate minimum
  fit.ridge(y, X, lsp[i.opt])
  ## return fit at minimum
} ## ridge
```

```
## test fun=I and inverse
n <- 10; A <- matrix(runif(n*n)-.5,n,n)
A <- A + t(A) ## make symmetric

B <- mat.fun(A) ## identity function case
if (max(abs(A-B))>norm(A)*1e-10) warning(" mat.fun I failure")

B <- mat.fun(A,function(x) 1/x) ## inverse function case
if (max(abs(solve(A)-B))>norm(A)*1e-10) warning("mat.fun inv failure")

A <- crossprod(A) ## ensure +ve def for sqrt case
B <- mat.fun(A,sqrt) ## sqrt function case
if (max(abs(A-B%*%B))>norm(A)*1e-10) warning("mat.fun sqrt failure")

## use try to check that an error is produced, as intended for non-square argument
if (!inherits(try(mat.fun(matrix(1:12,4,3)),silent=TRUE),"try-error"))
          warning("error handling failure")
```

Obviously you might choose to bundle these unit tests into a function that can then be called to run the tests, possibly returning a code indicating which error or none occurred, and/or printing warnings as appropriate.

Writing careful tests that are as comprehensive as possible is good practice, especially for code that will be used by others, and may be subject to future modification. But beware of the complacency that having good tests can bring with it. When modifying code, the mere fact that it still passes all its tests is no proof that the modification is correct. Also don't get in the habit of coding to the unit tests - debugging modifications until the unit tests are all passed and then assuming that all is OK.

A very common R error is for a function to accidentally use an object that just happened to be in your workspace: it then appears to work, although it will fail in general. Testing in a clean workspace is one defence. Another is also to check that your code still works when you change the names of all the objects used in testing (obviously deleting all the objects with the original names). Here is an illustration:

```
foo <- function(y) { tot <- 0; for (i in 1:n) tot <- tot + y[i] } ## buggy
n <- 100; y <- rnorm(n);foo(y) ## test fails to pick up problem
rm(n,y) ## test with new names and old objects removed
n2 <- 100; y2 <- rnorm(n2);foo(y2)
# Error in foo(y2) : object 'n' not found
```

## Exercise

Add a test for whether non symmetric matrices are being handled as designed.

```
n <- 10
A <- matrix(runif(n * n), n, n)
B <- try(mat.fun(A), silent = TRUE)
OK <- FALSE
if (inherits(B, "try-error")) { ## failed
  ## but need to check if it failed correctly by catching this case
  ## and producing appropriate error message...
  if (grep("matrix not symmetric", as.character(attr(B, "condition")))
== 1) OK <- TRUE
}
if (!OK) warning("non-symmetric matrix error handling failure")
```

## 12.4 Debugging

Debugging[19] is the process of finding and fixing errors in computer code. There are two aspects to debugging. Adopting the right approach, and using an appropriate debugger appropriately. Start with approach.

- Ideally we would not introduce bugs in the first place. But humans are error prone, and would probably not be creative if that was not the case. So recognise that you are error prone, by adopting working methods that minimize the chance of errors: design before you code. Make sure the task is broken down in a well structured manner, into functions that can be individually tested. Comment carefully, and test as you go along. Note that the act of describing what code is supposed to do in a useful comment is often a good way of noticing that it doesn't do that.
- Be aware that modification of code, for example to add functionality, is often the point at which bugs are introduced. Carefully commenting code to make its original function clear is the first line of defence against this. Being very careful to test modifications is also key.

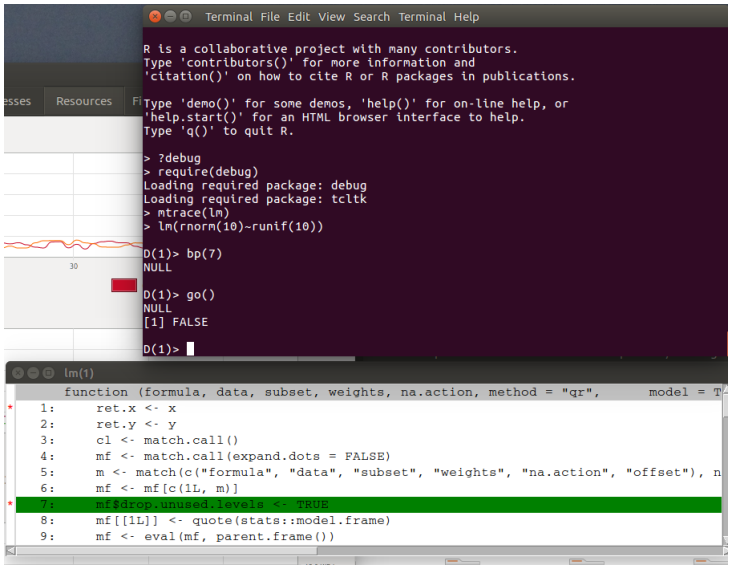
[19] The term comes from the early days of computing, when errors were sometimes caused by moths or other 'bugs' getting into the hardware of the computer. It stuck.

- But any interesting code will have bugs at some point in its development. Finding bugs usually requires that you adopt a scientific approach - gathering data on what might be wrong, forming hypotheses about what is causing the problems and then designing experimental tests to check this.
- Key to this approach is checking what your code is *actually* doing, and whether it conforms to what you think it *ought* to be doing.
- While going carefully through your code once looking for errors is sensible, staring at it for hours is rarely productive. You need to actively hunt down the errors. Sometimes this can be done by simply modifying the code to print out information as it runs, but more often the efficient way to proceed is to use a debugger to step through your code checking that it is producing what you expect at each step.

R provides `debug`, `debugcall`, `trace` and `traceback` functions for debugging, and Rstudio also has a basic debugger. However Mark Bravington's `debug` package is the most useful option currently available. It is not available on CRAN, so see section 1 for installation instructions. Use of `debug` is as follows:

1. Load the package into R with `require(debug)` or `library(debug)`.
2. Use `mtrace` to set which functions you want to debug. e.g. `mtrace(lm)` would set up debugging of R's own `lm` function. `mtrace(lm,FALSE)` turns off debugging of a specific routine (here `lm`), while `mtrace.off()` turns off all debugging.
3. To start debugging just call the function in the usual way. For example `lm(rnorm(10)~runif(10))`, will launch `lm` in debugging mode once we have used `mtrace` to set this up. See screenshot below.
4. Once in debugging mode the R prompt changes to `D(1)>` and a new window appears showing the source code of the function being debugged.
   - At the prompt you can issue R commands in the usual way. So setting or checking the values of variables is easy, for example.
   - Pressing the return key without an R command causes the next line of code of the function being debugged to be executed.
   - The command `go()` causes the function to run to completion, the first error, or to the next *breakpoint*.
   - Breakpoints are markers in the code indicating where you want execution to stop (to allow examination of intermediate variables, typically). They are set and cleared using the `bp()` function. For example:
     (a) `bp(101)` sets a breakpoint at line 101 of the current function, while `bp(101,FALSE)` turns it off again. The location of breakpoints is shown as a red star next to the appropriate code line in the source code window.
     (b) `bp(12,i>6,foo)` sets a breakpoint at line 12 of function `foo`, indicating that execution should only stop at this line if the condition `i>6` is met.

     See `?bp` for more details. Note that stepping over lengthy loops can be very time-consuming in the debugger (even one line loops), so setting a breakpoint after the loop and `go()`ing to it is a common use of breakpoints, even in simple code.
   - To step into a function called from the function you are debugging, either `mtrace` it, or set a breakpoint inside it.
   - `qqq()` quits the debugger from within a function being debugged, without completing execution of the remaining function code.
   - See `https://www.maths.ed.ac.uk/~swood34/RCdebug/RCdebug.html` for slightly more information, including how to deal with some minor niggles in `debug`.




## 12.5 Profiling

Once you have working tested debugged code you may, in some circumstances, want to work on its computational speed[20]. This can be especially important if coding statistical analyses of large data sets that may need to be repeated many times: for example the processing of medical images, satellite data, financial data, network traffic data, high throughput genetic data and so on.

The `Rprof` function in R profiles code by checking what function is being executed every `interval` seconds, where `interval` is an argument with default value 0.02, storing the results to a file (default `"Rprof.out"`). Function `summaryRprof` can then be used to tabulate how long R spent in each function. Two times are reported, total time in the function, and time spent executing code solely within the function, as opposed to other functions called by the function. The basic operation is as follows:

```
Rprof()
## the code you want to profile goes here
## only the calls to run the code are needed
## - not function definitions.
Rprof(NULL)
summaryRprof()
```

Here is a simple example.

```
> pointless <- function(A,B,y) {
+   C <- A%*%B%*%y  ## compute ABy
+   D <- solve(A,B)%*%y  ## compute A^{-1}By
+   E <- A + B
+   F <- A + t(B)
+ }
> n <- 2000; A <- matrix(runif(n*n),n,n)
> B <- matrix(runif(n*n),n,n); y <- runif(n)
> ## Profile a call to pointless...
> Rprof()
> pointless(A,B,y)
> Rprof(NULL)
> summaryRprof()
```

[20] you might also be concerned about its memory footprint, but we won't cover that here.

```
$by.self
                self.time self.pct total.time total.pct
"solve.default"      0.66    60.00       0.66     60.00
"%*%"                0.38    34.55       0.38     34.55
"t.default"          0.04     3.64       0.04      3.64
"+"                  0.02     1.82       0.02      1.82

$by.total
                total.time total.pct self.time self.pct
"pointless"           1.10    100.00      0.00     0.00
"solve.default"       0.66     60.00      0.66    60.00
"solve"               0.66     60.00      0.00     0.00
"%*%"                 0.38     34.55      0.38    34.55
"t.default"           0.04      3.64      0.04     3.64
"t"                   0.04      3.64      0.00     0.00
"+"                   0.02      1.82      0.02     1.82

$sample.interval
[1] 0.02

$sampling.time
[1] 1.1
```

Clearly `pointless` is spending most of its time in `solve` and `%*%`, which might encourage you to look again at these. Then recalling the discussions of matrix computation efficiency from section 11, you would modify the function to make it more efficient, and profile again.

```
> pointless2 <- function(A,B,y) {
+   C <- A%*%(B%*%y)  ## compute ABy
+   D <- solve(A,B%*%y)  ## compute A^{-1}By
+   E <- A + B
+   F <- A + t(B)
+ }
> Rprof()
> pointless2(A,B,y)
> Rprof(NULL)
> summaryRprof()
$by.self
                self.time self.pct total.time total.pct
"solve.default"      0.22    73.33       0.22     73.33
"t.default"          0.04    13.33       0.04     13.33
"%*%"                0.02     6.67       0.02      6.67
"+"                  0.02     6.67       0.02      6.67

$by.total
                total.time total.pct self.time self.pct
"pointless2"          0.30    100.00      0.00     0.00
"solve.default"       0.22     73.33      0.22    73.33
"solve"               0.22     73.33      0.00     0.00
"t.default"           0.04     13.33      0.04    13.33
"t"                   0.04     13.33      0.00     0.00
"%*%"                 0.02      6.67      0.02     6.67
"+"                   0.02      6.67      0.02     6.67

$sample.interval
[1] 0.02

$sampling.time
[1] 0.3
```