Stat 3701 Lecture Notes: Basics of R

Charles J. Geyer

January 23, 2023

> Can one be a good data analyst without being a half-good programmer? The short answer to that is, 'No.' The long answer to that is, 'No.'
>
> — Frank Harrell, 1999 S-PLUS User Conference, New Orleans (October 1999), quoted by the R function `fortune` in the CRAN package `fortunes`

1 License

2 R

2.1 Versions

- The version of R used to make this document is 4.2.1.
- The version of the `rmarkdown` package used to make this document is 2.19.

2.2 History

First came S, the first statistical computing language that was a real computer language. It was invented at Bell Labs by John Chambers and co-workers. Development started in 1975. It was first distributed outside of Bell Labs in 1980. Before 1988, S did not have real functions (it had macros to do the same job, but nowhere near as well). A version of S with real functions was released in 1988.

S always was and still is proprietary software. It is not free as in free beer nor free as in free speech. After purchasing it, one can only use it as the license allows. The name S is a one-letter name like C (another language from Bell Labs).

In 1988 another company was founded to market S, with additions, as a proprietary product called S-PLUS (or S+). After several changes of ownership, the current owner (TIBCO Software) does not seem to be selling it any more, at least not as a product by itself.

Then came R, an implementation of a language very like S. It was invented at the University of Auckland by Ross Ihaka and Robert Gentleman. The name R is a one-letter name like S but is also for Ross and Robert. Development started in 1992. The first stable release was in 2000. R is now developed and maintained by the "R Development Core Team."

R always was and still is free software. It is free as in free beer and free as in free speech. After downloading it for free, one can only use it as the license allows, but that license (GPL version 2) allows you to use it in any way you please and modify it in any way you please, subject only to the proviso that either you do not distribute your modifications outside of your organization or you distribute your modifications under the same license so that others are also free to use and modify your code as they please (subject to the same proviso).

One might think that whether software is free or not makes no difference to ordinary users because they don't want to modify it. But it makes a very big difference indirectly. If anyone anywhere in the world wants to modify R to solve a problem they have, they can do so, and they can make their solution available to everyone everywhere if they chose (and according to the license if they make it available to anyone, then they have to make it available to everyone). In contrast, proprietary software only gets features that the marketing department thinks will have a large enough number of users to make a profit. Since R is the only widely used free software statistical computing language, it has thousands of features that are not in any proprietary statistical computing language. So ordinary users may not want to modify R, but they benefit from the expert users who do want to.

According to an article in *IEEE Spectrum*, R is the fifth most popular computer programming language — not the fifth most popular *statistical* language but the fifth most popular language overall. Their top five were C, Java, Python, C++, and R in that order. R is (according to them) almost as popular as C++. The next most popular language that is vaguely similar is Matlab in fourteenth place (it is more a numerical analysis language than a statistics language). The next most popular statistics language is SAS in thirty-ninth place. No other statistical language is on their list, which ranks forty-nine languages.

Several proprietary companies have been built on R.

- In 2007 a company called Revolution Computing (later called Revolution Analytics, later sold to Microsoft) was founded to provide support for R and extensions for R specifically for big data. In July 2021, Microsoft announced that this product will be abandoned as a product and its technology will be replaced by the CRAN distribution of R and some new CRAN packages.
- In 2011 a company called RStudio (https://posit.co/) provided a product of the same name that is an integrated development environment for R (and they also provide other tools for use with R). RStudio is free software. In 2022 the company changed its name to Posit, which is supposed to show that is is also interested in Python and Visual Studio (Microsoft).

2.3 CRAN and Bioconductor

Everything R is found at CRAN (https://cran.r-project.org/) or Bioconductor (https://www.bioconductor.org/).

CRAN is where you get R if you don't already have it or if you need a new version. CRAN is also where more than ten thousand contributed extension packages for R come from (CRAN said "19104 available packages" when I wrote this sentence, no doubt there are more now). Any of these packages can be installed in seconds with one R command (or by mousing around in the menus in you are using an R app). These packages make R very powerful, able to do just about any statistical analysis anyone has ever thought of.

I have a habit of pronouncing CRAN as cee-ran like CTAN (https://www.ctan.org/) and CPAN (http://www.cpan.org/) after which CRAN is modeled. But almost everyone else pronounces it cran as in cranberry. I am trying to change, but sometimes forget.

Bioconductor is a smaller site that has many contributed extension packages for "bioinformatics" (it "only" has 2183 software packages as I write this sentence). Bioconductor packages are also easy to install, although not quite as easy as CRAN packages.

2.4 Manuals

All of the books that are the fundamental R documentation are free and found at CRAN (under the "Manuals" link in the navigation). We won't use any other books.

These books can also be found in the R installed on your computer. Run the R command `help.start()` and click on the link for the manual in the browser window that pops up. R tells you about this in its start-up message every time it starts.

The only one of the R manuals that is written for beginners (like you) is *An Introduction to R*. You should take a look at it. Anything that you find confusing in class or in the course notes (like this document) can probably be clarified by looking up the subject in *An Introduction to R.*

2.5 Other Books

There are now hundreds of books about some statistical topic using R. As I write, there are 61 books in the "Use R!" series from Springer, 40 books in the "R series" from Chapman & Hall/CRC, and (if I counted correctly) 130 books and videos with "R" in the title from O'Reilly. There are more from other publishers. Many of these books are very good, but we don't recommend any particular one for this course.

2.6 Why R?

R is the language of choice for statistical computing, at least among research statisticians and new companies (start-ups and companies that were recently start-ups like Google, Facebook, Amazon, and the like). The reasons are that R is a really developer friendly programming language. If anyone anywhere in the world wants to make an R package to do some problem, CRAN will make it available to everyone everywhere. And R does not make changes that break existing code (hardly ever). So such packages are easy to maintain. There is nothing like CRAN for competing proprietary statistical computing languages. Since R is free software, it is hackable by anyone who is a good programmer. It can be made to do anything. So again, if anyone anywhere in the world has solved your problem, there is likely a CRAN package to do what you want. Or if you are a company with great engineers, they can make R do whatever the company needs.

In older companies in which statistics has been important for many years (drug companies, for example) SAS may be what they use. They've been using it for decades and don't want to change. In other academic disciplines besides statistics, other computing languages may be used for statistics (SAS, SPSS, Stata, for example), but R is displacing them more and more. The reason is that there is so much available in R that is not available anywhere else. For example, the R package `aster` (http://cran.r-project.org/package=aster) does many forms of life history analysis that nothing else can do. So if hundreds of biologists want to analyze their data, they have to learn R. The ones that become good at it then become R advocates. Similar stories could be told about many kinds of statistical analysis that are available only in R.

You also hear Python, Perl, Julia, and other dynamic programming languages recommended for statistics or "data science" but none of these languages actually have much real statistics available, not even 1% of what is available on CRAN and Bioconductor. So they are useful if the statistics being done is very simple, but not otherwise.

But the main reason we are teaching you R is that is what we know. As a statistics major, you are doomed to learn R.

3 Some Very Basic Stuff

3.1 Style

Many organizations have "style guides" for the languages their programmers use. Many computer languages — all modern ones, including R — allow a great deal of latitude in how one writes. This can make it difficult for one programmer to read code written by another. So, just to make life easier for everyone, organizations make everyone use the same style.

There are three style guides for R that are somewhat authoritative. The first two are actual style guides, one by Google (https://google.github.io/styleguide/Rguide.xml) and another (http://adv-r.had.co.nz/Style.html), a modification of the Google guide by Hadley Wickham, author of the R packages `ggplot2`, `dplyr`, and many more (collectively called the Hadleyverse) and Chief Scientist at RStudio (https://www.posit.co/). The other is the source code of R by the R Development Core Team. (Go to https://cran.r-project.org/ and download the source code for the current version. I am not actually suggesting you do this now. Reading R source code is not easy. You will have to do that eventually to become an R guru, if you want to be one. But not yet.)

3.2 Commands

R commands do not need semicolons. R commands are terminated by newlines if they are syntactically complete. Otherwise they continue on the next line.

```
2 + 2
```

```
## [1] 4
```

```
2 +
2
```

```
## [1] 4
```

```
(2
+ 2)
```

```
## [1] 4
```

You can use semicolons to put a lot of R commands on one line, but don't (the Google R style guide recommends against this).

```
2 + 2; pi; "argle bargle"
```

```
## [1] 4
```

```
## [1] 3.141593
```

```
## [1] "argle bargle"
```

3.3 Assignment

Use `<-` for assignment, not `=`. Yes, the former is harder to type, but R from the most authoritative sources uses the former (including the R sources). The assignment operator for R and the S language that preceded it was `<-` for decades before John Chambers decided to add `=` to make the C++ programmers happy, which of course it didn't (actually I have no idea what his motivation was). The style guides and the R source code do not agree with Chambers.

Strangely, `->` is also an assignment operator. There is also a function `assign`. There are also `<<-` and `->>` operators.

```
foo <- 2
foo
```

```
## [1] 2
```

```
foo = 3
foo
```

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

```
4 -> foo
foo
```

```
## [1] 4
```

```
foo <<- 5
foo
```

```
## [1] 5
```

```
6 ->> foo
foo
```

```
## [1] 6
```

```
assign("foo", 7)
foo
```

```
## [1] 7
```

All of these do exactly the same thing in this context. In other contexts, they do different things. Inside function bodies, the assignments with the double headed arrows do something different, which we won't explain here. The `assign` function with optional arguments allows the most flexibility in assignment and the most precise specification of where the assignment is done, and we won't explain that here either.

In short, use `<-` for assignment.

3.4 Autoprinting

An R command that is not an assignment and is executed at the top level (not inside a function) automatically prints the value of the expression. So the way to see what an R object is, is just to make that object by itself an R command.

```
x <- 1
x
```

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

If the object is way too big to be worth looking at, the R functions `head`, `tail`, and `summary`, may be helpful.

```
x <- rnorm(10000)
head(x)
```

```
## [1]  1.1046535  0.1403364  1.1440854 -1.2598940 -0.8885096 -0.1871171
```

```
tail(x)
```

```
## [1]  1.04926524 -1.54409119  0.95152464 -0.62586385 -1.74670462  0.01909756
```

```
summary(x)
```

```
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max. 
## -4.180624 -0.677612  0.012015  0.008024  0.685807  3.636890
```

This feature is called autoprinting, and, though it seems very simple, it is actually rather complicated. It can be changed by using the R function `invisible`, which makes what would ordinarily print not print, and it actually works by hidden calls to the R function `print` or `show` (for S4 objects), which are generic functions, so what gets printed is appropriate for the type of object. That explanation may make no sense at this point, more about this later. Section 1.6 of *R Internals* explains (but that reference also makes no sense to beginners).

Summary: assignment commands don't print their values and other commands do.

This does not mean that assignment expressions don't have values. They do. That's why

```
x <- y <- 2
x
```

```
## [1] 2
```

```
y
```

```
## [1] 2
```

works.

3.5 Whitespace

As in C and C++, whitespace is mostly not needed in R. There are a few places where if things were run together, it would change the meaning, but not many. So one can write

```
x <- 2
x<-2
```

but the style guides recommend using white space to make it easier for people to read, whether the computer needs it or not.

3.6 Objects

> To understand computations in R, two slogans are helpful:
>
> - Everything that exists is an object.
> - Everything that happens is a function call.
>
> — John Chambers, quoted in Section 6.3 of *Advanced R* by Hadley Wickham

In R everything is an *object*. This is a triviality. The technical term used to refer to R thingummies is *object*.

3.7 Object-Oriented Programming

By calling thingummies "objects" this does not mean we are doing *object-oriented programming* (OOP) that you may have heard about in your C++ class. R objects are not like C++ objects. In R *everything* is an object. In C++ only very special things that are instances of C++ classes created with the C++ `new` operator are objects. In R, numbers, functions, bits of the R language (R expressions) are also objects. In C++ those things are definitely not objects. The C programming language doesn't have any objects at all, either in the C++ sense or the R sense.

R is not a particularly OOPy language. It does have three different OOP systems in the R core (the stuff you get without having to attach a package). And it has several more OOP systems in packages found in contributed extension packages on CRAN. But none of these OOP systems act like the C++ OOP system. Moreover none of the OOP systems are necessary to most R programming. Most R programming is not OOPy.

So forget what you learned about OOP in your C++ course. It won't help you with R.

3.8 Dynamic

R is a *dynamic* programming language. This means two things.

1. Any name can be assigned objects of any type. Unlike C and C++ you do not have to "declare" what type of object goes with each name.

2. You don't compile in the sense that you compile C and C++. You can just type or cut-and-paste R commands into the R program and see what happens.

This makes R infinitely easier to program than C/C++.

Nevertheless, R is just as powerful a language as C++. It is not as powerful as C. You can write operating systems (Windows, OS X, Linux) in C but not in any other so-called high level language. (That is why C is sometimes called a middle level language.)

3.9 Reproducibility

Although you can type or cut-and-paste R commands into the R program and see what happens, you shouldn't when doing serious work. Put all of your R in a file, and run it in a clean R environment. The command to do this from an operating system command line is

```
R CMD BATCH --vanilla foo.R
```

Here `foo.R` is the file with the R commands. A file called `foo.Rout` will be produced. All of the R commands in `foo.R` will be executed and all of the results will go into `foo.Rout`.

Except for randomness in random numbers used (and this can be removed by setting random number generator seeds, about which more later), this is 100% reproducible. Throwing code at the interpreter is not reproducible unless you deliberately start R so the global environment is empty and type in *exactly* the same commands every time. `R CMD BATCH` is much more reproducible. We will insist that assignments use `R CMD BATCH` or something else 100% reproducible (R markdown, for example).

An alternative to using a new R process started with the `--vanilla` flag is to put the command

```
freshr::freshr()
```

at the top of your R script or R markdown file. This will remove everything from the R global environment and unload any packages that are not loaded by default. So it has pretty much the same effect as starting a new R process.

3.10 Help

The R function `help` gives you the somewhat inappropriately named help on a subject (usually on a function or on a dataset, but help on other things also exists). Many people do not find this help helpful. It depends on what you are looking for.

R help pages are modeled after unix man pages. They are supposed to be complete, concise, and correct. Most computer "help" outside of R help pages or unix man pages is none of these things. It tries to be helpful and friendly and thus omits lots of things (not complete), belabors what it does cover (not concise), and dumbs down things to the point of being wrong (not correct). Many people nevertheless find this stuff helpful. Again, it depends on what you are looking for.

R help pages, on the other hand, do correctly answer all questions you might have. But sometimes the answers are so terse, you may not understand them. They are thus useful only when you know the basics of the subject but need to know some technical detail that you either forgot or never new. *Exactly* what does this function do? *Exactly* what is the name of some argument to this function? Questions like that. R help pages do not give any background. They do not tell you why or when you might want to use a function.

For background, you need a book, perhaps *Introduction to R*, perhaps some other book. In between help pages and books, are package vignettes, which not all packages have. For example, in the CRAN package `mcmc` (http://cran.r-project.org/package=mcmc) there are several vignettes; the main one that shows basic use of the package is shown by the R commands

```
library(mcmc)
vignette("demo", "mcmc")
```

(We cannot show the output of this command in this document because R displays the output in a separate window. But if you do this interactively, it will work. This applies to many commands in this section.)

As an example of using help, I had to do `help(vignette)` because I can never remember whether this function is named `vignette` or `vignettes` or even how to spell it. Moreover, I did not remember the order of the arguments or their names, so I needed the help for that too.

To see all the vignettes for this package do

```
library(mcmc)
vignette(package = "mcmc")
```

There is another command `help.search` that may be useful when you do not know the name of the function (or other thing) that you want help for. For example

```
help.search("help")
```

returns a list of R functions that have “help” in their help page titles or names. Some of these are involved in the R help system (including `help.search` itself); others are not.

R has two shorthands for getting help.

```
?whatever
??whatever
```

are shorthand for

```
help(whatever)
help.search(whatever)
```

(The reason for the lack of quotation marks in these examples is explained in the following section).

3.11 Non-standard Evaluation

Some R functions use what some call “non-standard evaluation,” (a term apparently coined by Hadley Wickham (http://adv-r.had.co.nz/)). Instead of just evaluating their arguments, they play tricks with some of them.

This “playing tricks” is possible because R expressions are R objects too. So R can look at them and try to understand them and not just evaluate them.

An example of this is when you do a plot and do not specify labels for the horizontal and vertical axes, R makes up some labels from the expressions that specify the horizontal and vertical coordinates of the points plotted.

Another example is help.

```
help("ls")
help(ls)
?ls
?"ls"
```

all do the same thing (show the help for the R function named `ls`). But

```
help("function")
help(function)
?function
?"function"
```

do not all do the same thing. The ones without quotation marks are an error and an incomplete R statement, neither of which shows the help page. What is going on here?

The non-standard evaluation here is inside the R function named `help`. Before that function is even invoked, the R interpreter (the part of the R program that actually executes R commands) must parse the command. The text `help(function)` is not a valid R statement because `function` is a keyword that must be followed by a function definition. So an error occurs before the R function named `help` can try to do its trick of working whether or not there are quotation marks. The problem is similar with `?function` except here since nothing follows `function` the R interpreter realizes that you could follow this with a definition of a function, so it just prints the prompt for a continuation line. And, of course, this has nothing to do with what you wanted (help for the function named `function`).

The example of the R help system, shows that non-standard evaluation is sometimes problematic. When in doubt, use the quotation marks.

A lot of R is like this. R does a lot of tricky things that are helpful most of the time (omitting quotation marks saves two whole keystrokes when getting help) but when it is not helpful, it is *seriously* unhelpful.

In programming, one should never use non-standard evaluation unless that is what has to be done to get the job done. For example, inside a function you are writing, you may need to assure that a package is loaded, The function for this is named `require`. And it plays non-standard evaluation tricks just like `help`. Both

```r
require(mcmc)
require("mcmc")
```

work. But you should only use the latter in programs you write.

The ability of R to do computing on the language is remarkable and far beyond what most computer languages can do (only LISP and Scheme are comparable). So the fact that R can do non-standard evaluation is a good thing. The fact that non-standard evaluation can be used where not helpful does not automatically make it bad. Only that particular use is (arguably) bad. It probably should not have been used in the R functions `help` and `help.search`, but it is too late to change that now.

4 Functions

4.1 Functional Programming

R is a *functional* programming language. This means that functions are first class objects that you can do anything with that you could do to any other object. You can assign them to variable names, or you can assign them to be parts of compound objects (like lists), or you can use them as arguments or values of other functions.

In this one respect, R is exactly like Haskell or Javascript or F# or Clojure or Ruby or other functional programming languages. In other aspects, R is very different from those languages.

> This [closures] is the best idea in the history of programming languages.
>
> — Douglas Crockford

Crockford was talking about Javascript, but R has the same idea. It is the function named `function`, which creates functions. Technically, they are called “closures” for reasons to be explained later.

```r
typeof(function(x) x)
```

```
## [1] "closure"
```

In this respect R is different from and better than S. The R function named `function` creates true closures, where S and S-PLUS did not, as explained in Section 3.3.1 of the R FAQ.

4.2 Pure Functional Programming

In a *pure* functional programming language, functions do not have side effects. If called with the same values for the same arguments, they must always produce the same results.

There are very few totally pure functional programming languages because they cannot vary what they do to adjust to the outside world. They cannot do input or output, they cannot do random numbers, they cannot tell the time or date. And so forth. R isn’t totally pure either.

But it is more or less pure as far as computation goes. Unlike C and C++ functions, R functions cannot change the values of their arguments. In R there is only one kind of argument (R object). There is none of this nonsense about pointers and references. In C, you can write a function that changes its argument (or more precisely what its argument points to)

```c
void bar(int *x)
{
    *x = 2;
}
```

when you call it as follows

```
int one = 1;
bar(&one);
```

after the call to `bar` returns, the value of `one` is 2.

In C++ the above also works (because almost all C is also valid C++) but the following also works

```
void baz(int& x)
{
    x = 2;
}
```

when you call it as follows

```
int one = 1;
baz(one);
```

after the call to `baz` returns, the value of `one` is 2.

In R nothing like this is possible the way R functions are commonly programmed.

```r
ralph <- function(x) {
    x <- 2
    return(x)
}
x <- 1
ralph(x)
```

```
## [1] 2
```

```r
x
```

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

Inside the function, the value of `x` is changed to 2, but outside the function the value of `x` is unchanged by the function. There are two variables named `x` one inside the function and one outside. This is just like C or C++ functions that have non-pointer, non-reference arguments. The C or C++ function

```
int ralph(int x)
{
    x = 2;
    return x;
}
```

behaves just like our R function of the same name above.

R environments and R reference classes, which are syntactic sugar over environments, are exceptions. But they are little used. (At least they are rarely used explicitly. A new environment is involved in every function definition and every function invocation. More on this later.)

The vast majority of R functions cannot change their arguments the way C and C++ functions can. This makes R much easier for programmers to program, and, more importantly, *much easier for users to use*. The only way an ordinary R function that does not do I/O or use random numbers can have an effect on the world outside itself is to return a value.

Every time a function that does not do input, use random numbers, or access the date, time, process ID, or some such thing (all of which are forms of input) is called with the same values of the same arguments, it will return the same result. In short, it is *pure functional*. Thus *pure functional* is the normal style of R programming. Use it, don't try to fight it.

If you need to return lots and lots of stuff, then put it in a list (which makes it one R object), and return that. That's what model fitting functions like `lm` and `glm` do. It is a very common R design pattern. More on this later.

4.3 Functions Assigned Names

Here are some examples of R being a functional programming language.

```
fred <- function(x, y) x + y
fred(2, 3)
```

```
## [1] 5
```

Never mind that this is just a toy example (we don't need a function for addition, we already have one). This shows the idiomatic way to define a function in R, at least a one-liner. A function that needs multiple lines of code for its implementation needs curly brackets around the body of the function

```
fred <- function(x, y) {
   x <- cos(x)
   y <- exp(y)
   x + y
}
fred(2, 3)
```

```
## [1] 19.66939
```

But unlike in C and C++, the curly brackets are not needed for one-liners.

How does the function know what value to return? It returns the value of the last expression it evaluates, in the examples above the value of `x + y`.

The R function `return` immediately returns a value from a function (and no further code in the function body is executed). So we could have written

```
fred <- function(x, y) return(x + y)
fred(2, 3)
```

```
## [1] 5
```

if we wanted our function to look more like C/C++. But it still doesn't really look like C or C++ because in them `return` is a keyword (part of the syntax of the language) not a function, so (in C/C++) one could write `return x + y` without parentheses. In R that is invalid.

Unlike in C or C++, an R function cannot *not* return a value. Many R functions seem to not return a value, but they do.

```
sally <- plot(1:10, 2:11)
sally
```

```
## NULL
```

If a function cannot think of anything better to return, it can return `invisible(NULL)`. That means it returns the R object `NULL` and that object does not get printed when the value is not assigned so users don't see it and can imagine that it doesn't exist.

But every function returns a value. Many R functions that ordinary users think don't return values (because they never see them and don't wonder about them) actually return objects with useful stuff in them. We won't worry about that now. Just keep it in mind: every R function returns a value (which is an R object).

Just one more example illustrating that the R function `return` returns immediately:

```r
fred <- function(x, y) {
   x <- cos(x)
   y <- exp(y)
   return(x + y)
   cat("just something here that can never get executed\n")
}
fred(2, 3)
```

```
## [1] 19.66939
```

4.4 Functions as Arguments to Functions

R functions can be arguments to other R functions.

4.4.1 R Functions that work on mathematical functions.

- R functions `grad` and `hessian` and `jacobian` in R package `numDeriv` differentiate R functions (approximately) and return the values of the derivatives evaluated at specified points. These are vector or matrix valued derivatives of vector-to-scalar or vector-to-vector functions.
- R function `integrate` does numerical integration of a mathematical function provided to it as an R function.
- R functions `nlm` and `optim` and `optimize` and many other R functions that do optimization, optimize a mathematical function provided to them as R functions.
- R function `uniroot` finds a root (zero) of a univariate mathematical function provided to it as an R function.
- R function `boot` in R recommended package `boot` (installed by default in every installation of R) approximately simulates the sampling distribution of any estimator provided to it as an R function applied to any data using Efron's nonparametric bootstrap.
- R function `metrop` in CRAN package mcmc simulates the distribution of any continuous random vector whose log unnormalized probability density function can be provided to it as an R function.

So this is a very widely used R design pattern. To tell an R function about a mathematical function, provide it an R function that evaluates that mathematical function (and that function should be pure).

4.4.2 R Functions of the Apply Family

R function `apply` and friends (`eapply`, `lapply`, `mapply`, `rapply`, `sapply`, `tapply`, and `vapply` in R package `base` and `mclapply` and `clusterApply` in R package `parallel`) apply an arbitrary function (provided as an R function) to each element of some compound object (vector, list, data frame, table, etc.) and the ones in R package `parallel` do each instance in a different parallel process (if possible).

4.4.3 R Functions of the Higher-Order Family

In computer science, any function that takes a function as an argument or returns a function as a value is called a *higher-order function*. So all of the functions in the preceding two sections are higher-order functions. But this terminology is not widely used in R.

In languages much newer than R a bunch of terminology for higher-order functions has grown up, and R has copied it. See `help("funprog")` for the list. The most widely used are `Map`, `Reduce`, and `Filter`. Some of these are built on functions of the apply family.

4.4.4 Some Other Similar R functions

In this section, we put other functions that take functions as arguments like `sweep`, `aggregate`, `outer`, and `kronecker`, just because we don't know where else to put them.

4.4.5 Example

The R function `optimize` optimizes functions of one (scalar) variable. Its first argument is the function to optimize, and the second is an interval over which to minimize it.

```r
fred <- function(x) exp(x) - log(x)
optimize(fred, c(0.1, 10))
```

```
## $minimum
## [1] 0.5671468
##
## $objective
## [1] 2.330366
```

If we look at a plot done by the following code

```r
curve(fred, from = 0.1, to = 10, log = "y")
```

which is

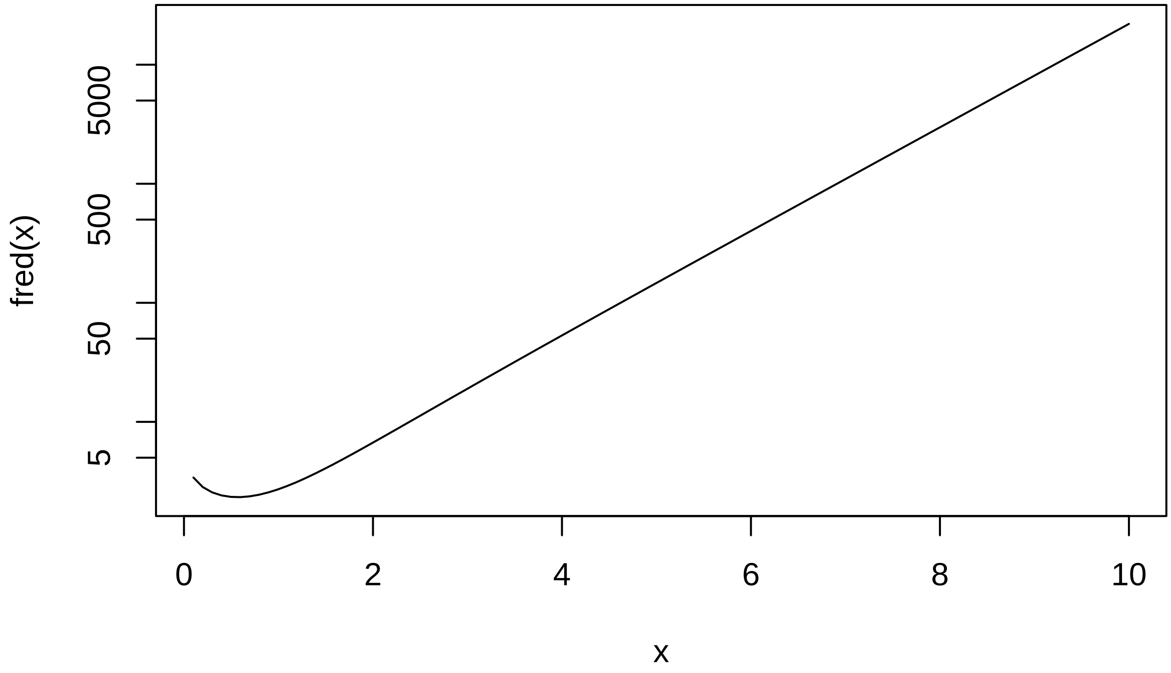

Figure 1: Graph of Mathematical Function Computed by R Function fred

we can see that `optimize` seems to have found the solution. We used a log scale for the vertical axis because otherwise we wouldn't be able to see where the minimum was (try the same without `log = "y"`).

4.5 Anonymous Functions

R functions don't need names. The R function whose name is `function` makes functions. They don't need to be assigned to names to work. If we don't bother to give a function a name, the jargon widely used in functional programming is that we are using an *anonymous function*. We can redo the preceding example using an anonymous function.

```
optimize(function(x) exp(x) - log(x), c(0.1, 10))
```

```
## $minimum
## [1] 0.5671468
##
## $objective
## [1] 2.330366
```

This is just the preceding example with the expression defining `fred` plugged in where `fred` was passed as an argument to `optimize`. In general, anyplace you can use an R object, you can also use an expression defining that object (except when nonstandard evaluation is involved).

You don't need to do things like this very often,

```
(function(x) exp(x) - log(x))(2.7)
```

```
## [1] 13.88648
```

but it does work. Here an anonymous function is being evaluated at the argument value 2.7. The parentheses around the function definition are necessary to make the whole function definition something we can apply the function invocation parentheses to.

4.6 Functions Returning Functions as Values

This is not a common R design pattern, but a few examples exist.

R function `ecdf` returns a function that evaluates the empirical cumulative distribution (corresponding to a data vector).

The term *function factory* is applied to functions whose job is to make functions, like `ecdf`. Another example is the function `make.logl` in Section 7.5.4 below.

Another aspect of this subject is deferred to Section 6.2 below.

5 Vectors

In R all of the objects that "hold stuff" are vectors. There are no objects that can hold only one number or only one character string. You may think there are, but those objects are really vectors.

```
one <- 1
one
```

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

```
is.vector(one)
```

```
## [1] TRUE
```

```
length(one)
```

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

Why does R print

```
[1] 1
```

and

```
[1] TRUE
```

for these results? (The `##` in front of output is a foible of the R package `knitr` and also of `rmarkdown` which uses `knitr` that I am using to make this document. It doesn't appear when you are using R yourself.)

If a vector takes many lines to print, each line starts with the index of the first component of the vector to appear on that line.

```
1:40
```

```
##  [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 31 32 33 34 35 36 37 38 39 40
```

As we can see, the R binary operator `:` makes sequences. As we can also see, R uses one-origin indexing like FORTRAN rather than zero-origin indexing like C and C++.

R vectors come in two kinds. In *atomic vectors* all of the components have to have the same type. In *lists* the components can be of different types.

5.1 Types

The types an R atomic vector can have are (table from Section 2.1.1 of the *R Language Definition*)

typeof	mode	storage.mode
logical	logical	logical
integer	numeric	integer
double	numeric	double
complex	complex	complex
character	character	character
raw	raw	raw

The column headings are the names of R functions (`typeof`, `mode`, and `storage.mode`) and the entries are what they say about various kinds of atomic objects. There is one other R function of this kind, `class` that also gives useful information about R objects.

5.1.1 Numeric

```
one <- 1
typeof(one)
```

```
## [1] "double"
```

```
mode(one)
```

```
## [1] "numeric"
```

```
storage.mode(one)
```

```
## [1] "double"
```

```
class(one)
```

```
## [1] "numeric"
```

The result of `storage.mode` is what C and C++ think of as the type. Type `"double"` means it is stored as a C or C++ `double`.

One might ask, why is R storing the integer 1 as a double? And the answer is sometimes it does. For the most part R users (even knowledgeable users) do not need to distinguish between integers and doubles (between "fixed point" and "floating point" numbers). The R functions `mode` and `class` don't distinguish, reporting `"numeric"` for both.

Let's try another.

```
lone <- length(one)
typeof(lone)
```

```
## [1] "integer"
```

```
mode(lone)
```

```
## [1] "numeric"
```

```
storage.mode(lone)
```

```
## [1] "integer"
```

```
class(lone)
```

```
## [1] "integer"
```

For some reason, the R function `length` returns its result as numbers of type `"integer"`. So we see R really does have two kinds of numbers. But, as we said above, most users never notice the difference and don't need to notice.

This is different from C and C++ which have an insane assortment of numbers. They have three floating point types (`float`, `double`, and `long double`) and I don't know how many integer types (`char`, `short`, `int`, `long`, and `long long`, and all of these with `unsigned` in front, like `unsigned char`, and maybe some I forgot about). In R we don't worry about any of this. For the most part R numbers are just numbers, and we won't say any more about them until we discuss computer arithmetic.

5.1.2 Complex

The `"complex"` type is what you'd expect.

```
typeof(sqrt(-1))
```

```
## Warning in sqrt(-1): NaNs produced
```

```
## [1] "double"
```

Hmmmmm. That didn't work. Let's try again.

```
typeof(sqrt(-1+0i))
```

```
## [1] "complex"
```

What did they do?

```
sqrt(-1)
```

```
## Warning in sqrt(-1): NaNs produced
```

```
## [1] NaN
```

```
sqrt(-1+0i)
```

```
## [1] 0+1i
```

Apparently, R only thinks that $\sqrt{-1} = i$ when you tell it that you are working with complex numbers by specifying -1 as a complex number, `-1+0i` in R notation. Otherwise, it says that square roots of negative numbers do not exist. The result `NaN` stands for “not a number”. We will learn more about it when we get to computer arithmetic. This behavior is useful in statistics, most of the time, because we don’t use complex numbers much.

5.1.3 Logical

In R the two logical values `TRUE` and `FALSE` are special values different from any numbers. `TRUE` and `FALSE` are not R objects but keywords. The technical term in R for words that are part of the R syntax and cannot be variable names is “reserved word” rather than “keyword”. To see the documentation on them, do `?Reserved` or `help(Reserved)`.

R, for backwards compatibility with its predecessor S, also understands `T` and `F` as synonyms, but you should never use them. The difference is that `T` and `F` are not keywords but just R objects that can be redefined, and that will wreak havoc if you are expecting them to behave as logical values.

```
typeof(TRUE)
```

```
## [1] "logical"
```

```
typeof(T)
```

```
## [1] "logical"
```

```
T <- "spaghetti"
typeof(T)
```

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

If you try

```
TRUE <- "spaghetti"
```

```
## Error in TRUE <- "spaghetti": invalid (do_set) left-hand side to assignment
```

you will find that you can’t do that. `TRUE` is a keyword and cannot be the name of an R object.

This is very different from ancient C and C++ which did not have logical types. Instead they used zero or types convertible to zero like null pointers as their equivalent of `FALSE`, and everything else as their equivalent of `TRUE`. Modern C and C++ do have what they call “Boolean” types, they are new and not used by all programmers.

Always use `TRUE` and `FALSE`. Never use `T` and `F` instead.

5.1.4 Character

Type `"character"`, also called “string” by some (probably because of the influence of C and C++), we have already seen. Here is another example.

```
LETTERS
```

```
##  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S"
## [20] "T" "U" "V" "W" "X" "Y" "Z"
```

5.1.5 Raw

Type `"raw"` cannot be constructed in R. It is there for the use of C or C++ functions called from R. They can return objects of type `"raw"` that R does not understand and cannot use. They can only be passed to C or C++ functions called from R that understand them. So we won’t pay any more attention to this type.

5.2 Atomic Vectors

We have already seen atomic vectors (since R has no scalars, every object that may have looked like a scalar is really an atomic vector). So nothing more needs to be said except that R vectors are much smarter than C vectors. Like C++ objects of class `std::vector` they know their own size and type. (In general, every R object knows lots of things about itself). Unlike C++ vectors, R vectors don't have methods. Instead there are functions that work on them. (This is because all of this goes back before R to S before it had OOP.)

```
length(LETTERS)
```

```
## [1] 26
```

```
head(LETTERS)
```

```
## [1] "A" "B" "C" "D" "E" "F"
```

```
tail(LETTERS)
```

```
## [1] "U" "V" "W" "X" "Y" "Z"
```

```
length(colors())
```

```
## [1] 657
```

```
head(colors())
```

```
## [1] "white"         "aliceblue"     "antiquewhite"  "antiquewhite1"
## [5] "antiquewhite2" "antiquewhite3"
```

```
tail(colors())
```

```
## [1] "yellow"      "yellow1"     "yellow2"     "yellow3"     "yellow4"
## [6] "yellowgreen"
```

```
length(1:20)
```

```
## [1] 20
```

```
length(double())
```

```
## [1] 0
```

R is just fine with vectors of length zero.

5.3 Vectors, Operators, and Functions

Since every R object that holds stuff (not functions, expressions, and the like) is a vector, operators and functions have to deal with that. Since S and R have had this "everything is a vector" from their beginnings, every function and operator knows how to deal with vectors (at least each knows its own special way to deal with vectors).

5.3.1 Componentwise with Recycling

Many mathematical functions deal with vectors the same way. If the operands or arguments are vectors of the same length, they work componentwise.

```
1:5 + 6:10
```

```
## [1]  7  9 11 13 15
```

```
1:5 * 6:10
```

```
## [1]  6 14 24 36 50
```

```
1:5 ^ 6:10
```

```
## Warning in 1:5^6:10: numerical expression has 15625 elements: only the first
## used
```

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

Oops! That last wasn't what we meant.

```
(1:5)^(6:10)
```

```
## [1]       1     128    6561  262144 9765625
```

And many functions work the same

```
dnorm(1:5, 6:10, 11:15)
```

```
## [1] 0.03270787 0.03048103 0.02849997 0.02673528 0.02515888
```

If we look up what this function does (`?dnorm`), we find that it calculates the probability density function of normal distributions; it calculates $f_{\mu,\sigma}(x)$, where the arguments to `dnorm` are x, μ, and σ in that order. The call above is the same as

```
dnorm(1, 6, 11)
```

```
## [1] 0.03270787
```

```
dnorm(2, 7, 12)
```

```
## [1] 0.03048103
```

```
dnorm(3, 8, 13)
```

```
## [1] 0.02849997
```

```
dnorm(4, 9, 14)
```

```
## [1] 0.02673528
```

```
dnorm(5, 10, 15)
```

```
## [1] 0.02515888
```

except for being five commands rather than one.

When the arguments are different lengths, these operators and functions still work. When one of the arguments is length one, these do what you'd expect.

```
1:5 + 2
```

```
## [1] 3 4 5 6 7
```

```
1:5 * 2
```

```
## [1]  2  4  6  8 10
```

```
(1:5)^2
```

```
## [1]  1  4  9 16 25
```

```
dnorm(1:5, 2, 2)
```

```
## [1] 0.1760327 0.1994711 0.1760327 0.1209854 0.0647588
```

This is a special case of a more general rule called the *recycling rule*.

- If the operands or arguments are vectors of different lengths, then the length of the result is the length of the longest operand or argument.
- Each operand or argument is extended to this length by recycling: for example an operand `1:2` is extended to length 5 as the vector with components 1, 2, 1, 2, 1.

The recycling rule can be very confusing.

```
dnorm(1:2, 1:3, 1:5)
```

```
## [1] 0.39894228 0.19947114 0.10648267 0.09666703 0.07820854
```

```
dnorm(1, 1, 1)
```

```
## [1] 0.3989423
```

```
dnorm(2, 2, 2)
```

```
## [1] 0.1994711
```

```
dnorm(1, 3, 3)
```

```
## [1] 0.1064827
```

```
dnorm(2, 1, 4)
```

```
## [1] 0.09666703
```

```
dnorm(1, 2, 5)
```

```
## [1] 0.07820854
```

Sometimes R gives a warning about this kind of confusion.

```
1:2 + 1:3
```

```
## Warning in 1:2 + 1:3: longer object length is not a multiple of shorter object
## length
```

```
## [1] 2 4 4
```

and sometimes it doesn't (as we saw above with `dnorm`).

I don't recommend you use confusing recycling. If you do, put in a comment to explain.

5.3.2 Reduce

There is another common design pattern where functions take a vector and produce a number

```
sum(1:5)
```

```
## [1] 15
```

```
prod(1:5)
```

```
## [1] 120
```

```
max(1:5)
```

```
## [1] 5
```

```
min(1:5)
```

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

There is even a higher-order function `Reduce` that uses this design pattern with an arbitrary function.

```
Reduce("+", 1:5, 0)
```

```
## [1] 15
```

```
Reduce(function(x, y) if (x > y) x else y, 1:5, -Inf)
```

```
## [1] 5
```

(the first does the same as `sum`, the second does the same as `max`).

5.3.3 Ifelse

Since every R object is a vector (except for those that aren't). One should use this feature of the language as much as possible. In R, iteration (`for` and `while` loops) is used much less than in C and C++. A lot of iteration can be replaced by functions that operate on vectors (as all R functions have no choice but to do).

For example, `sum` does sums without a loop, and `Reduce` does this for any operation. The functions in the apply family (listed in Section 4.4 above allow the design pattern of componentwise calculation with recycling to be applied with arbitrary functions.

The R function `ifelse` allows the if-then-else design pattern to be applied vectorwise.

```
x <- rnorm(6)
x
```

```
## [1] -0.3449863  0.6815931  0.3055691 -2.0791415 -0.7505709  1.8501364
```

```
ifelse(x < 0, x - 3, x + 3)
```

```
## [1] -3.344986  3.681593  3.305569 -5.079141 -3.750571  4.850136
```

```
x + 3 * sign(x)
```

```
## [1] -3.344986  3.681593  3.305569 -5.079141 -3.750571  4.850136
```

5.4 Indexing

R has four (!) different ways to "index" vectors (extract subvectors, or otherwise refer to subvectors).

- vector of positive integers
- vector of negative integers
- logical vector
- character vector

Indexing is also sometimes called subscripting. Because vector indices are often denoted as subscripts in mathematics. For example, `?Subscript` gives the help for the indexing operators, but so does `?Extract`, which is the official name of this help page (`Subscript` is an "alias").

5.4.1 Indexing with Positive Integers

```
LETTERS[7]
```

```
## [1] "G"
```

But here as everywhere else what would be a scalar in C or C++ can be a vector in R

```
LETTERS[2 * 1:7 - 1]
```

```
## [1] "A" "C" "E" "G" "I" "K" "M"
```

So that illustrates the first type (positive integer vector).

5.4.2 Indexing with Negative Integers

```
LETTERS[- (2 * 1:7 - 1)]
```

```
##  [1] "B" "D" "F" "H" "J" "L" "N" "O" "P" "Q" "R" "S" "T" "U" "V" "W" "X" "Y" "Z"
```

(This would be different if we omitted the parentheses.) Negative indices indicate components to omit. The result is all of the components except the ones whose indices are the positive integers corresponding to these negative integers.

5.4.3 Index Zero?

```
LETTERS[0:5]
```

```
## [1] "A" "B" "C" "D" "E"
```

Apparently zero is just ignored in an integer index vector.

5.4.4 Indexing with Logicals

```
LETTERS[seq_along(LETTERS) %% 2 == 1]
```

```
##  [1] "A" "C" "E" "G" "I" "K" "M" "O" "Q" "S" "U" "W" "Y"
```

The logical expression indicates odd indices. We see a bunch of new stuff here. The R function `seq_along` makes the index set for a vector, although we could replace this with

```
LETTERS[1:length(LETTERS) %% 2 == 1]
```

```
##  [1] "A" "C" "E" "G" "I" "K" "M" "O" "Q" "S" "U" "W" "Y"
```

The `%%` operator calculates remainders (from division, in this case division by 2). As in C and C++, the operator for logical equality is `==`.

We could also do this more simply with

```
LETTERS[seq(1, length(LETTERS), 2)]
```

```
##  [1] "A" "C" "E" "G" "I" "K" "M" "O" "Q" "S" "U" "W" "Y"
```

but that wouldn't illustrate logical indexing.

5.4.5 Indexing with Character Strings

The fourth kind of indexing (with a character vector) doesn't work unless the R object we are taking components out of has names. So let's give our example names.

```
names(LETTERS) <- letters
LETTERS[c("d", "o", "g")]
```

```
##   d   o   g
## "D" "O" "G"
```

The R function `c` "concatenates" (pastes together) vectors to make one vector.

Here we see a very powerful feature of R. Some functions can appear on the left-hand side of an assignment. Actually (technical quibble!), the function being called in the example above is not named `names` but rather is named `names<-`

```
get("names<-")
```

```
## function (x, value)  .Primitive("names<-")
```

but most R users don't know that and don't need to know that. They just "know" that `names` can be used on either side of an assignment.

The R documentation encourages this way of thinking. From the help page for these functions, which one gets by `?names` or `help(names)`,

```
Usage:

     names(x)
     names(x) <- value
```

Indexing also works on the left-hand side of an assignment.

```
LETTERS[seq_along(LETTERS) %% 2 == 1] <- "woof"
LETTERS
```

```
##      a      b      c      d      e      f      g      h      i      j      k
## "woof"    "B" "woof"    "D" "woof"    "F" "woof"    "H" "woof"    "J" "woof"
##      l      m      n      o      p      q      r      s      t      u      v
##    "L" "woof"    "N" "woof"    "P" "woof"    "R" "woof"    "T" "woof"    "V"
##      w      x      y      z
## "woof"    "X" "woof"    "Z"
```

The R objects `LETTERS` and `letters` are, like `T` and `F` and `pi` and some other things, already defined when you start R (all of the predefined constants, have the same help page, so asking for the help for any one of them shows all of them).

When we started redefining `LETTERS` we made a new variable in our global environment (what R calls where it keeps objects you give names by assignment). But the objects with the same names defined by R are still there hidden by our assignments (they are in another place, more on this later). We can see them again by removing ours.

```
rm(LETTERS)
LETTERS
```

```
##  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S"
## [20] "T" "U" "V" "W" "X" "Y" "Z"
```

This is the magic of R environments. More on this later.

5.5 Type Coercion

R, like C and C++ does a lot of *type coercion* (changing variable of one type into another behind your back). Since R has fewer types than C and C++, its type coercion is (slightly) less crazy than C/C++'s.

The most useful is coercion of logical to numeric in arithmetic contexts. For example,

```
x <- rnorm(100)
sum(x < 0)
```

```
## [1] 58
```

counts how many components of `x` are negative. Section 2.4 of *Introduction to R* explains this.

Sometimes useful and sometimes not is automatic coercion of all kinds of objects to character in character contexts.

```
foo <- 1:10
names(foo) <- foo + 100
foo
```

```
## 101 102 103 104 105 106 107 108 109 110
##   1   2   3   4   5   6   7   8   9  10
```

```
class(names(foo))
```

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

```
letters[1:3] <- rnorm(3)
letters[1:10]
```

```
##  [1] "0.460543445646067" "0.927733107548232" "-1.49917692970779"
##  [4] "d"                 "e"                 "f"
##  [7] "g"                 "h"                 "i"
## [10] "j"
```

The first is arguably useful. The latter may do more harm than good. The character conversions are explained in Section 2.6 of *Introduction to R*.

Coercions can also be performed by explict use of functions.

```
as.character(1:5)
```

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

```
as.numeric(letters[1:5])
```

```
## Warning: NAs introduced by coercion
```

```
## [1]  0.4605434  0.9277331 -1.4991769         NA         NA
```

```
as.logical(0:5)
```

```
## [1] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
```

Here my claim that R help pages are complete, concise, and correct is wrong (I reported this as a bug, but the R core team did not agree, so the bug was not fixed). The information is in `?Logic` but apparently nowhere else. That help page says

> Numeric and complex vectors will be coerced to logical values, with zero being false and all non-zero values being true.

The help page for `as.logical` does not say what it does when the argument is numeric. It appears to be following the way of C and C++, converting zero to `FALSE` and everything else to `TRUE` except for `NA` or `NaN` which are left as (logical) `NA`.

```
foo <- as.integer(c(0:5, NA))
foo
```

```
## [1]  0  1  2  3  4  5 NA
```

```
as.logical(foo)
```

```
## [1] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE    NA
```

```
foo <- c(0:5, pi, NA, NaN)
# has to be type double if contains pi
foo
```

```
## [1] 0.000000 1.000000 2.000000 3.000000 4.000000 5.000000 3.141593       NA
## [9]      NaN
```

```
as.logical(foo)
```

```
## [1] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE    NA    NA
```

5.6 Lists

Vectors that are not atomic can hold different types of R object. They are created by the R function named `list`.

```
sally <- list(color = "red", number = pi,
    method = function(x, y) x + y)
sally
```

```
## $color
## [1] "red"
##
## $number
## [1] 3.141593
##
## $method
## function(x, y) x + y
```

And they are generally called lists.

5.7 More On Indexing

With lists there are two more kinds of indexing. All of what we saw in Section 5.4 above works on lists. But for lists there is also another entirely different kind, or perhaps two entirely different kinds, depending on how you count.

```
sally[3]
```

```
## $method
## function(x, y) x + y
```

makes a list whose single component is the third component of `sally`. That is often not what we want, so there is another kind of indexing using double square brackets.

```
sally[[3]]
```

```
## function(x, y) x + y
```

See the difference?

- `sally[3]` is a list having one element named `method`, which is a function.
- `sally[[3]]` is that function.

Hence `sally[3]` is equal to `list(sally[[3]])`.

The reason why we need both kinds (single square brackets and double square brackets) is that you usually (?) want what double square brackets does, but that doesn't always work.

```
sally[2:3]
```

```
## $number
## [1] 3.141593
##
## $method
## function(x, y) x + y
```

```
sally[[2:3]]
```

```
## Error in sally[[2:3]]: subscript out of bounds
```

I didn't know what would happen with the latter until I tried it, but the documentation says double square brackets indexing can only select a single component (the argument must be a vector of length one). Since these are operators the documentation is a bit hard to find. You need to quote bits of syntax to see the documentation. For double square brackets `?"[["` or `help("[[")` does the job. The same goes for other reserved words: `?"function"` or `help("function")` is the way to see the documentation for the function named `function`.

One can also use the `$` operator instead of the `[[` operator if the list has names and if the name in question is a valid R name (also called symbol). `?name` shows the documentation for that, except it doesn't tell you what the valid names are. For some strange reason, `?make.names` does tell you what the valid names are.

So

```
sally$method
```

```
## function(x, y) x + y
```

```
sally[["method"]]
```

```
## function(x, y) x + y
```

```
sally[[3]]
```

```
## function(x, y) x + y
```

all do the same thing.

But if we change the example

```
names(sally)[3] <- "function"
sally[["function"]]
```

```
## function(x, y) x + y
```

```
sally[[3]]
```

```
## function(x, y) x + y
```

but

```
sally$function
```

```
## Error: unexpected 'function' in "sally$function"
```

`sally$function` is an error because `function` is an R reserved word hence not a valid name (this is not an Rmarkdown code chunk because `knitr` (which underlies `rmarkdown`) is too clever for its own good here and cannot just echo the actual R error message and has to make up its own error message, which is worthless).

As

```
sally
```

```
## $color
## [1] "red"
##
## $number
## [1] 3.141593
##
## $`function`
## function(x, y) x + y
```

suggests,

```
sally$`function`
```

```
## function(x, y) x + y
```

does work, but IMHO this is less clear than `sally[["function"]]`.

5.8 Nit Picking about Indexing

As with `+`, indexing syntax is syntactic sugar for function calls.

```
get("[")
```

```
## .Primitive("[")
```

```
get("[[")
```

```
## .Primitive("[[")
```

```
get("$")
```

```
## .Primitive("$")
```

```
get("[<-")
```

```
## .Primitive("[<-")
```

```
get("[[<-")
```

```
## .Primitive("[[<-")
```

but, like most R users, you don't need to know these functions exist unless you want to implement them yourself for some R class that you have created, but before we can understand what that means we would have to learn about the basic R OOP system (so-called S3 classes), and this is not the time for that.

Nevertheless, this does illustrate the quotation in Section 3.6 above: in R *everything that happens is a function call,* even things that don't look at all like function calls.

6 More On Functions

The examples in Section 4 above illustrate most of the things that knowledgeable users of R (what this course is trying to turn you into) do with functions. But there is lots more to learn about functions.

6.1 Storing Functions in Objects

One can put an anonymous function into a compound object.

```
sally <- list(color = "red", number = pi,
    method = function(x, y) x + y)
sally$method(2, 3)
```

```
## [1] 5
```

As we saw in Section 5.7 above, one R syntax to extract a named element from a list is the `$` operator. The C++ syntax to extract a method or a field from an object is the `.` operator. Since both languages are about the same age, there is no reason to think the C++ one is in any way better, even though it is more widely copied in other languages. This sort of looks like `sally` is a C++-like object and `method` is a method of its class. But R isn't using any such notions here. An R list can contain anything. So it can contain functions. Here `sally$method` happens to be a function. To invoke a function, you put its arguments in a comma-separated list in round brackets after it. Like the example above.

Here is another example that shows the same thing with weirder syntax.

```
sally[[3]](2, 3)
```

```
## [1] 5
```

Here sally[[3]] is the third component of sally, which is a function. We invoke it in the usual way.

6.2 Functions whose Values are Functions

Here is an example of a function returning a function.

```
fred <- function(y) function(x) x + y
fred(2)(3)
```

```
## [1] 5
```

Now we have gotten to the power of functional programming that makes it both much more powerful than procedural programming languages like C++ and Java but also very confusing to programmers trained in procedural programming. What is going on here?

Here fred is a function whose value is a function. So when we say fred(2) we have invoked the function fred with the argument 2 and gotten a function of one argument that adds 2 to that argument. (One might prefer saying it adds y to that argument, where y is the value of the argument of fred. Here y is 2.) So fred(2) is a function of one variable, that adds 2 to its argument. We invoke it by putting its argument in parentheses after it. Hence the above.

All very logical. Mr. Spock or Lieutenant Commander Data would have no trouble. You may have some trouble following this.

Functions like this are used in various places in mathematics. We can think of this fred as almost but not quite the same thing as our first example of a function named fred in Section 4.3 above. They do the same thing. Both “are” functions of two arguments that add their arguments, but “are” has to be in scare quotes because only the first example is really a function of two arguments. It is invoked fred(2, 3) just like any other R function of two arguments. The higher-order version is a function of one argument that returns a function of one argument, so it is invoked fred(2)(3) as above. It is almost but not quite the same. The two functions do the same thing, but they are not invoked in the same way. Going from one to the other is called currying and uncurrying.

This design pattern is less widely used in R than the other kind of higher-order function (taking functions as arguments). But there are a few examples. The R function ecdf, given a data vector, returns the empirical cumulative distribution function (ECDF) corresponding to that data. What it returns is an R function that evaluates the ECDF. The R functions D and deriv differentiate R expressions

```
D(expression(x^3), "x")
```

```
## 3 * x^2
```

```
D(expression(sin(x)), "x")
```

```
## cos(x)
```

```
deriv(expression(sin(x)), "x")
```

```
## expression({
##     .value <- sin(x)
##     .grad <- array(0, c(length(.value), 1L), list(NULL, c("x")))
##     .grad[, "x"] <- cos(x)
##     attr(.value, "gradient") <- .grad
##     .value
## })
```

but deriv can also produce a function

```
deriv(expression(sin(x)), "x", function.arg = "x")
```

```
## function (x)
## {
##     .value <- sin(x)
##     .grad <- array(0, c(length(.value), 1L), list(NULL, c("x")))
##     .grad[, "x"] <- cos(x)
##     attr(.value, "gradient") <- .grad
##     .value
## }
```

Readers may find these examples using deriv incomprehensible at this point. Hopefully, these will make sense after we learn more.

6.3 Partially Evaluated Functions

Many, perhaps most, knowledgeable R users do not understand functions that return functions, currying, and uncurrying. They do not need to because these techniques are not much used in R. But they do understand something very similar.

```
fred <- function(x, y) x + y
fran <- function(x) fred(x, 2)
fran(3)
```

```
## [1] 5
```

Here we have a function fred with two arguments x and y, but we want to consider it a function of x only for fixed y. We call this new function fran. This is a very useful technique. It is widely used in mathematics.

Name the mathematical functions that correspond to the R functions fred and fran by the letters f and g, respectively (because in math we usually denote functions by single letters). What is the relationship between f and g? It is

$$g(x) = f(x, 2), \qquad \text{for all } x.$$

This concept is so important that mathematicians have many ways to write it. We could also say that the function g is $f(\,\cdot\,, 2)$. We could also say that the function g is $x \mapsto f(x, 2)$.

6.4 Some Real Nit Picking

6.4.1 The Function Function

We said above the R function named function makes other functions, but that is not quite correct. If function were actually an R function then typing the function name on a line by itself would tell R to print the definition of the function. This works with every other function, for example,

```
sum
```

```
## function (..., na.rm = FALSE)  .Primitive("sum")
```

But it doesn't work with function: if you type function on a line by itself at the R interpreter, it just says this is not a complete command and gives you a continuation prompt. So function is actually an R keyword (part of the syntax) rather than just the name of a function that is defined somewhere in R. There is actually an R function named function, and it is called when expressions containing the keyword function are evaluated. To see it we can do either of the following

```
get("function")
```

```
## .Primitive("function")
```

```
`function`
```

```
## .Primitive("function")
```

This doesn't tell us much, only that the R function named function is very low level. There is almost no R code in its definition.

6.4.2 The Addition Function

Here is another definition of our first example that is very hard to read.

```
fred <- `+`
fred(2, 3)
```

```
## [1] 5
```

This tells us that in R *almost everything* is a function. A lot of R syntax is syntactic sugar for function calls. When you write `2 + 3` in R, what actually happens is that a function whose name is `+` and which has two arguments is invoked with 2 and 3 as the arguments.

This illustrates again the quotation in Section 3.6 above: in R *everything that happens is a function call,* even things that don't look at all like function calls.

6.4.3 The Quit Function

Why does one type `q()` to quit R if you are using the command line rather than a GUI app? Because the function named `q` is the function that quits R and `q()` is calling this function with no arguments.

In R lots of things are functions that are not functions in other programming languages.

7 Still More On Functions

Before you can consider yourself a knowledgeable R user, you have to know about

- named arguments,
- default values for arguments,
- missing arguments,
- `...` (the bit of R syntax that enables variable number of arguments)

7.1 Named Arguments

Like in C and C++ every function argument in R has a name. How else would you refer to it inside the function? Unlike in C and C++ the names of R function arguments can be used outside the function when invoking the function. Here's an example.

```
fred <- function(x, y) x^y
fred(2, 3)
```

```
## [1] 8
```

```
fred(3, 2)
```

```
## [1] 9
```

```
fred(x = 2, y = 3)
```

```
## [1] 8
```

```
fred(y = 3, x = 2)
```

```
## [1] 8
```

Again, never mind that we don't need a function to do exponentiation because we already have the caret operator to do it. This is just a toy function that unlike our earlier `fred` is not a symmetric function of its arguments.

When we don't use names and reverse the order of the arguments, the value of the function changes. When we do use names, the order doesn't matter. The names tell R which argument is which, and the order is ignored. This means users don't have to remember the order so long as they remember (or look up in the documentation) the names.

R has a feature called *partial matching* of argument names (done by the R function `pmatch`). Users don't have to specify the whole argument name, just enough to uniquely specify the argument.

```r
fred <- function(xerxes, yolanda) xerxes^yolanda
fred(y = 3, x = 2)
```

```
## [1] 8
```

7.2 Default Values for Arguments

In an R function arguments can be given *default* values. If the user omits the argument, then the default is used.

```r
fred <- function(xerxes = 4, yolanda = 5) xerxes^yolanda
fred()
```

```
## [1] 1024
```

```r
fred(2)
```

```
## [1] 32
```

```r
fred(2, 3)
```

```
## [1] 8
```

```r
fred(yola = 3)
```

```
## [1] 64
```

Default values for arguments can be complicated expressions.

```r
fred <- function(x, fun1 = mean, fun2 = function(x) mean(x^2))
    fun2(x - fun1(x))
fred(1:10)
```

```
## [1] 8.25
```

The above definition is so confusing that I had trouble reading it a week after I wrote it. The first line of the function definition is the signature (what the arguments are) and the second line is the body. This function calculates `fun2(x - fun1(x))`.

By default `fred(x)` calculates the variance of the empirical distribution for data `x` (dividing by n instead of $n - 1$). But by using the optional arguments we can calculate the median absolute deviation from the median

```r
fred(1:10, median, function(x) median(abs(x)))
```

```
## [1] 2.5
```

Default values for arguments can also depend on other arguments. My first attempt at writing the function above was

```r
fred <- function(x, mu = mean(x), fun = function(x) mean(x^2))
    fun(x - mu)
fred(1:10)
```

```
## [1] 8.25
```

This shows that the default value for the second argument (`mu`) can depend on the first argument (`x`). The third argument does not depend on the first argument because in `function(x) mean(x^2))` the name `x` is the argument of the anonymous function this expression creates. Unfortunately, this makes a bad example because this version of the function is harder to use, so we wouldn't actually write it this way.

```r
fred(1:10, median(1:10), function(x) median(abs(x)))
```

```
## [1] 2.5
```

For what seems to be the ultimate in R default values trickery, let us look at the R function `svd`

```r
svd
```

```
## function (x, nu = min(n, p), nv = min(n, p), LINPACK = FALSE)
## {
##     if (!missing(LINPACK))
##         stop("the LINPACK argument has been defunct since R 3.1.0")
##     x <- as.matrix(x)
##     if (any(!is.finite(x)))
##         stop("infinite or missing values in 'x'")
##     dx <- dim(x)
##     n <- dx[1L]
##     p <- dx[2L]
##     if (!n || !p)
##         stop("a dimension is zero")
##     La.res <- La.svd(x, nu, nv)
##     res <- list(d = La.res$d)
##     if (nu)
##         res$u <- La.res$u
##     if (nv) {
##         if (is.complex(x))
##             res$v <- Conj(t(La.res$vt))
##         else res$v <- t(La.res$vt)
##     }
##     res
## }
## <bytecode: 0x5624f346e120>
## <environment: namespace:base>
```

The defaults for arguments `nu` and `nv` depend on `n` and `p` which don't exist when the function is invoked. They aren't created until lines 5 and 6 of the function body. But this works because R does *lazy evaluation* of function arguments. They are not evaluated until they are used, and `nu` and `nv` are not used until line 9 of the function body, which is after `n` and `p` are initialized. All of this is explained, more or less, in the documentation (`?svd`).

7.3 Missing Arguments

Arguments can be missing whether or not there is a default value. But if so, the function must either never try to use them or define them itself. And why would it want to do either of these? That would be like not having the argument at all.

But the R function `missing` allows us to do different things in the function depending on whether the argument is missing or not. Here is an example

```r
sample
```

```
## function (x, size, replace = FALSE, prob = NULL)
## {
##     if (length(x) == 1L && is.numeric(x) && is.finite(x) && x >=
##         1) {
```

```
##         if (missing(size))
##             size <- x
##         sample.int(x, size, replace, prob)
##     }
##     else {
##         if (missing(size))
##             size <- length(x)
##         x[sample.int(length(x), size, replace, prob)]
##     }
## }
## <bytecode: 0x5624f28f6638>
## <environment: namespace:base>
```

It is clear that we have to use `missing` rather than a default argument in order to do different things in case the first argument has length one or not.

Unfortunately, this is a horrible example of really bad programming. R does this to be backwards compatible with S, and S did it in an extremely misguided attempt to be helpful. But this is the kind of help users don't need. It is very surprising. Only the most expert of users can remember this weirdness, and they don't always remember it. Better the original programmer of `sample` had never thought of this trick. This is an example you should not emulate!

There are many examples in the R code base where `missing` is used well. But they are complicated, and we don't want to explain them now.

7.4 `...`

The `...` syntax allows R functions to have an arbitrary number of arguments. For example,

```
args(list)
```

```
## function (...)
## NULL
```

The `...` matches every argument. This shows how you can capture the `...` arguments if one wants to operate on them in a function you are writing

```
alice <- function(...) {
    args <- list(...)
    if (is.null(args$fred)) args$fred <- "J. Fred Muggs"
    args
}
alice(x = "foo", y = "bar")
```

```
## $x
## [1] "foo"
##
## $y
## [1] "bar"
##
## $fred
## [1] "J. Fred Muggs"
```

```
alice(fred = 10)
```

```
## $fred
## [1] 10
```

This is the design pattern you use when you want to set a `...` argument if the user has not set it, and otherwise want to leave it the way the user set it. We also see that the R idiom for testing whether a list does not have a named element is `is.null(args$fred)`.

Whether partial matching is used on arguments that are not `...` arguments depends on where the `...` is: named arguments that come before `...` are partially matched, and those that come after `...` are not partially matched. This is so hard to remember that many help pages, for example `optimize`, explicitly say

> Note that arguments after `...` must be matched exactly.

```
fred <- function(..., herman = "default")
    list(dots = list(...), herman = herman)
fred(g = 1:10, h = c("red", "orange", "yellow", "blue",
    "green", "indigo", "violet"), i = function(x) x)
```

```
## $dots
## $dots$g
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## $dots$h
## [1] "red"    "orange" "yellow" "blue"   "green"  "indigo" "violet"
##
## $dots$i
## function(x) x
##
##
## $herman
## [1] "default"
```

Argument `herman` is not matched by argument `h` because `herman` comes after `...` and so must be exactly matched.

But

```
fred <- function(herman = "default", ...)
    list(dots = list(...), herman = herman)
fred(g = 1:10, h = c("red", "orange", "yellow", "blue",
    "green", "indigo", "violet"), i = function(x) x)
```

```
## $dots
## $dots$g
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## $dots$i
## function(x) x
##
##
## $herman
## [1] "red"    "orange" "yellow" "blue"   "green"  "indigo" "violet"
```

Now argument `herman` is matched by argument `h` because `herman` comes before `...` and so may be partially matched.

There are probably many uses of `...` that I have never thought of. The two main uses are

- to allow the function to work on a variable number of arguments (like the R function `list` and many other R functions), and
- to pass arguments to another function (like the R function `optimize` and many other R functions).

From ?optimize, in the “Usage” section

```
optimize(f, interval, ..., lower = min(interval),
         upper = max(interval), maximum = FALSE,
         tol = .Machine$double.eps^0.25)
optimise(f, interval, ..., lower = min(interval),
         upper = max(interval), maximum = FALSE,
         tol = .Machine$double.eps^0.25)
```

(R allows either American or British spelling for this function name) and later on in the “Arguments” section

```
...: additional named or unnamed arguments to be
         passed to ‘f’.
```

7.5 A Long Example (Maximum Likelihood Estimation)

7.5.1 Make Up Data

Suppose we want to do maximum likelihood estimation for the gamma distribution with unknown shape parameter and known scale parameter, which we take to be the R default value.

First we make up data.

```r
alpha <- pi
n <- 30
set.seed(42)
x <- rgamma(n, shape = alpha)
```

Here we first make up the true unknown parameter value alpha and sample size n. Of course we actually know alpha but the whole point of the example is to pretend we don’t know alpha and have to estimate it from the data x (which is a random sample from the distribution with this parameter value).

The reason for the set.seed command is so that we get the same x every time this document is created. If we deleted that statement, we would get a different x every time.

7.5.2 Using Dot-Dot-Dot

First define the log likelihood function. The argument we want to optimize over (the parameter) has to come first if it is to be optimized by R function optimize (or R function optim or R runction nlm if this is a vector argument rather than a scalar argument).

```r
logl <- function(alpha, x)
    sum(dgamma(x, shape = alpha, log = TRUE))
```

R function optimize needs an interval in which it is to seek the optimum. Here we choose the interval mean plus or minus 3 standard deviations, which should contain the true unknown mean with high probability (and for the gamma distribution with default scale parameter alpha is the mean). Except, when the lower endpoint of this interval is negative that is outside the range of the variable, that won’t work. So make the lower endpoint a positive number much smaller than the mean in this case.

```r
interval <- mean(x) + c(-1, 1) * 3 * sd(x)
interval <- pmax(mean(x) / 1e3, interval)
interval
```

```
## [1] 0.00307587 8.44529568
```

So now we are ready to do the maximization.

```r
oout <- optimize(logl, maximum = TRUE, interval, x = x)
oout$maximum
```

```
## [1] 3.04927
```

```
mean(x)
```

```
## [1] 3.07587
```

The point of showing both the maximum likelihood estimator (MLE) and the sample mean, both of which are consistent and asymptotically normal estimators of the unknown parameter α, is just to show that they are different.

A plot of the log likelihood done by the following code

```
mylogl <- Vectorize(function(alpha) logl(alpha, x))
curve(mylogl, from = interval[1], to = interval[2],
    xlab=expression(alpha), ylab=expression(logl(alpha)))
```

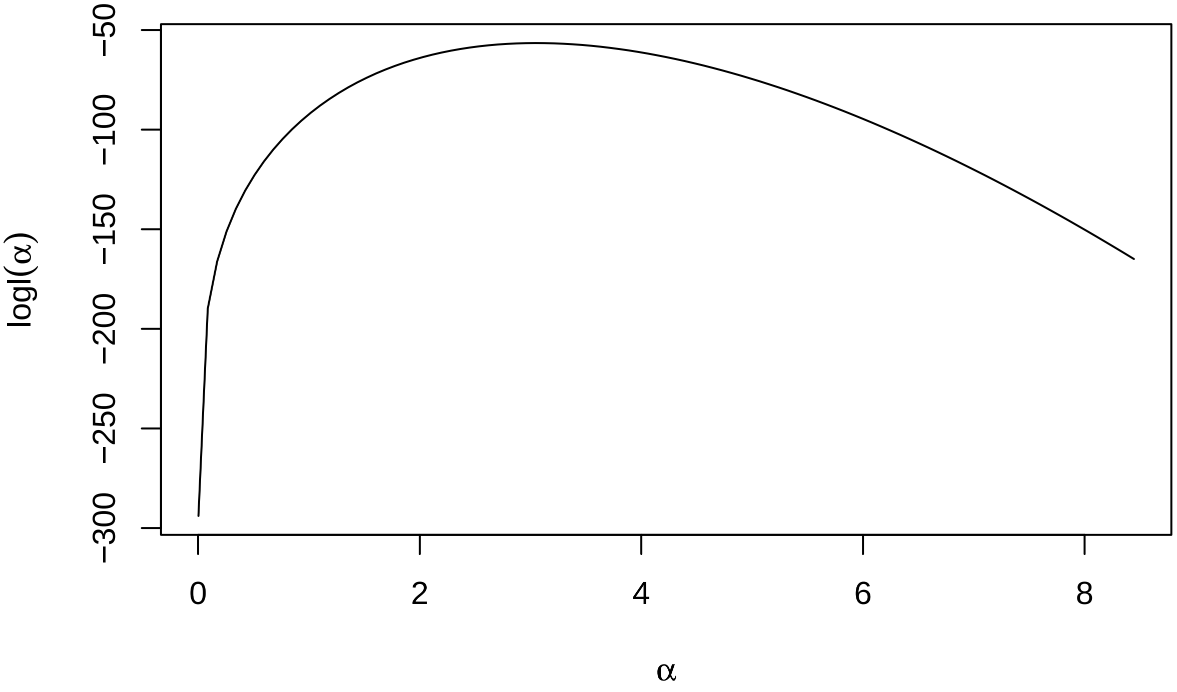

Figure 2: Graph of Log Likelihood

shows that the optimization seems to have worked. The tricks needed to draw this curve we do not want to explain right now.

All of that is interesting, and we will go over it in detail at some point in the course. But in this section, the only point that is interesting is how we are using the `...` argument to `optimize`. The R function `optimize` does not have an argument named `x` or even an argument that comes before `...` in the argument list that can be partially matched to `x` (from the documentation quoted above only the arguments `f` and `interval` come before `...` and neither begins with `x`). Thus `optimize` considers `x` a `...` argument and passes it to `f` when `f` is called (many times) from inside `optimize` to evaluate the function being maximized (that is `logl` which is called `f` inside `optimize`). When `optimize` calls `f` it does it by defining an anonymous function

```
function(arg) f(arg, ...)
```

(typing `optimize` at the R command line shows you its definition) and since we know that in this case `...` matches only the argument `x = x`, this is the same as defining the objective function to be

```
function(arg) f(arg, x = x)
```

or, since `f` is another name for `logl`, as

```
function(arg) logl(arg, x = x)
```

Note that here the R function `optimize` is using the trick of "partially evaluated functions" explained in Section 6.3 above. It takes the given function, which it calls `f` and which can have many arguments, and converts it to an anonymous function of one argument. It passes this to a C function named `do_fmin` to actually do the optimization, so we cannot see how that works without reading the C source code for R, which we won't bother with. The point is that this C function only needs to know how maximize R functions of one variable. It doesn't need to know about any other variables.

7.5.3 Alternative Solution (Using Global Variables)

The preceding section shows the approved (in some circles) way to do that problem.

But here is another way (that some people deem evil and stupid). Just define `logl` as a function of one variable.

```
logl <- function(alpha)
    sum(dgamma(x, shape = alpha, log = TRUE))
```

and then do the optimization as before except now we omit the `x = x`.

```
oout <- optimize(logl, maximum = TRUE, interval)
oout$maximum
```

```
## [1] 3.04927
```

How does that work? How does `logl` when called from within `optimize` find out what `x` is?

The short answer is that it looks it up in the R global environment (which is where we defined it in the first place). So it works. And we didn't need `x = x`.

And now for the caution about this method. Global variables are evil. (An interesting bit of computing history: Wiki Wiki Web was the first Wiki that Wikipedia and zillions of other wikis copy (Wikipedia entry Wiki)).

In serious work, global variables should never be used. Especially, they should never be used in code that you make for others to use. What if you call the data `x` inside your function and the user calls the data `y` outside your function. That won't work. But if the `...` trick is used, then the user can still call the data `y` and make the argument to match `x` via the `...` mechanism `x = y`, that is, the argument named `x` (in the function you wrote) is the data named `y` (outside the function) by the user.

So put `logl` back the way it was originally

```
logl <- function(alpha, x)
    sum(dgamma(x, shape = alpha, log = TRUE))
```

and now call the data `y`

```
y <- x
rm(x) # now x is gone
```

and

```
oout <- optimize(logl, maximum = TRUE, interval, x = y)
oout$maximum
```

```
## [1] 3.04927
```

still works.

In short, *don't use global variables.* Except. The Perl slogan is TIMTOWTDI (there is more than one way to do it), pronounced tim-toady (Wikipedia entry). This could also be an R slogan. There is no one true way to use R. There are many ways of R. As we said above, the way of this problem that uses global variables is the simplest, easiest, and most R-ish for one-off uses when the programmer and the user are one.

You only need to avoid global variables to be politically correct in the computer science sense (as the Wiki Wiki Web page cited above explains) or to have your code usable by others (including your future self six months from now).

7.5.4 Having Your Cake and Eating It Too (Closures)

There is a way that combines the virtues of both of the preceding ways. No global variables, and no `...` variables either. But it is somewhat mysterious.

It uses the fact that R functions are closures (mentioned in Section 4.1 above). They remember local variables in the environment in which they were created. So we can put `x` in there.

This is an example of what Hadley Wickham calls the *function factory* design pattern, as mentioned in Section 4.6 above.

Here's how that works. For ease of explanation, we do it in two steps. Note that this is also very similar to our toy example in Section 6.2 above except that this is a non-toy function.

```
make.logl <- function(x) function(alpha)
    sum(dgamma(x, shape = alpha, log = TRUE))

logl <- make.logl(y)
```

and then do the optimization as before when we could omit the `x = y`.

```
oout <- optimize(logl, maximum = TRUE, interval)
oout$maximum
```

```
## [1] 3.04927
```

It works the same. But how does it work?

- R function `make.logl` is a higher order function. It is a function that returns a function, and that function has the same definition as the R function `logl` in the other examples.
- The returned function, like any R function, "knows" local variables in the environment in which it was defined, which is the execution environment of R function `make.logl` when it is invoked.
- The only local variable there is named `x`.
- When R function `make.logl` is invoked, we give it the value named `y` in the R global environment.
- Inside `make.logl` and *inside the function that `make.logl` returns* this value has the name `x`.
- So this R function `logl` works the same way all the others did.

Note that we are really using the power of functional programming here.

- R function `optimize` is a higher order function because its first argument is a function.
- R function `make.logl` is a higher order function because it returns a function.

- So we are using a higher order function to create the function that we pass as an argument to another higher order function.

7.5.5 The Same Except Even More Mysterious

As always, we can use an anonymous function instead of giving it the name `make.logl`.

```
logl <- (function(x) function(alpha)
    sum(dgamma(x, shape = alpha, log = TRUE)))(y)
```

and then do the optimization as before

```
oout <- optimize(logl, maximum = TRUE, interval)
oout$maximum
```

```
## [1] 3.04927
```

The reason for the parentheses around the anonymous function definition

```
function(x) function(alpha)
    sum(dgamma(x, shape = alpha, log = TRUE))
```

is to make the whole definition something to which R can apply the other use of parentheses: function invocation, in this case `(y)`.

But this is uncomprehensible to almost all R users. Until you get quite used to this design pattern, it is perhaps best to do it as in the preceding section.

7.5.6 Where is X?

But where does R put `x`? In the environment of the function.

```
environment(logl)$x
```

```
##  [1] 5.3396900 1.8035110 2.7199928 1.1513247 2.4719243 1.6810323 6.9406407
##  [8] 2.5406468 1.1712249 0.8665140 5.2110715 4.4706090 4.2691549 3.4834492
## [15] 4.4443983 1.4835538 2.1667205 6.7240825 6.1847368 3.2562157 3.1559991
## [22] 2.0398708 0.5531137 1.3241273 3.4336934 3.9113621 4.5918039 1.7446038
## [29] 1.6205851 1.5204572
```

```
identical(environment(logl)$x, y)
```

```
## [1] TRUE
```

The local variable `x` is stored in the function itself.

8 Stop Me before I Crash and Burn

There is a very old computer slogan GIGO (garbage in, garbage out) (Wikipedia entry) that goes back to the 1950's. There is a more modern meaning *garbage in, gospel out* that refers to people believing anything that comes from a computer no matter how ridiculous (the internet is never wrong).

Another R slogan could be *garbage in, error messages out* (GIEMO) R functions should not allow users to shoot themselves in the foot. They should not be *footguns*. This is unlike some languages we know

> Within C++, there is a much smaller and cleaner language struggling to get out.
>
> — Bjarne Stroustrup
>
> C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off.

— Bjarne Stroustrup

Since so much of R is functions, if none of the functions are footguns, then the whole language is not a footgun (almost). Of course, some R functions are footguns, such as the notorious `sample` mentioned above (Section 7.3) and the notorious `[` function also mentioned above (Section 5.4) but without mentioning its footgun behavior, which involves matrices (and will be explained in the next handout). But the vast majority of R functions are not footguns because they follow GIEMO. The first job of a good R function that is usable by humans who make mistakes is to check that no errors are possible. In order to not be a footgun our log likelihood function in Section 7.4.1 above should have been something like

```
logl <- function(alpha, x) {
    stopifnot(is.numeric(alpha))
    stopifnot(is.finite(alpha))
    stopifnot(length(alpha) == 1)
    stopifnot(alpha > 0)
    stopifnot(is.numeric(x))
    stopifnot(is.finite(x))
    stopifnot(x > 0)
    sum(dgamma(x, shape = alpha, log = TRUE))
}
```

Of course, since `dgamma` is not a footgun, you can rely on it to catch some of these errors. But you should not. You should catch them yourself, for the following reasons.

- Just from looking at our code, we can tell our function is not a footgun.
- If we look at the R code for `dgamma`, we see that it hardly checks anything and instead leaves these checks for the C function it calls to do the work.
- Reading the C source code is hard.
- Actually, a lot more than hard. We just don't want to go there.

Let's check that it still works.

```
oout <- optimize(logl, maximum = TRUE, interval, x = y)
oout$maximum
```

```
## [1] 3.04927
```

9 Control-Flow Constructs

9.1 If-Then-Else

R has loops and if-then-else, but they behave quite differently from C and C++.

For one thing, like everything else in the R language, they are expressions. We have already seen this in Section 5.3.2 above where we used `if (x > y) x else y` as part of a larger expression. You can't do that in C or C++.

9.2 For

For another thing `for` loops in R iterate over the elements of a vector or list. They do not do arithmetic on indices. In this they work more like so-called for-each loops in Java or C++ (more on this below).

Thus, while the C or C++ or Java

```
for (int i = 0; i < n; i++) {
    // do case i
}
```

does have an R equivalent

```
for (i in 1:n) {
    # do case i
}
```

the R is different in that it does not do any arithmetic on `i` and `1:n` is just a vector of integers like any other place `1:n` is used in R.

R also has the following. Suppose `todo` is a list of some sort(s) of R objects, then

```
for (o in todo) {
    # process object o
}
```

iterates over elements of this list. This is what we mean by the `for` control-flow construct in R being more like for-each in Java or C++ (and like nothing in C).

Beginners who have been brainwashed in languages like C or C++ are tempted to treat `for` in R like `for` in C or C++, writing the last example with indices

```
for (i in 1:length(todo)) {
    o <- todo[[i]]
    # process object o
}
```

This works, of course, but it is clumsy and inelegant. Don't use indices unless you have to, for example, when processing several vectors of the same length in parallel.

Just for our own amusement (readers can skip to the end of this subsection and not miss anything about R) we look at the Java and C++ for-each code. In Java if `todo` is an object of a class that implements the `Iterable<Foo>` interface, then

```
for (Foo o : todo) {
    // process object o
}
```

does the same thing as the R for loop above. The C++ analog looks just the same except that the description of what `todo` has to be in order for this to work is so complicated that I cannot figure out any way to condense it here.

9.3 Repeat

Because R does not have the C and C++ kind of `for`, the R `for` cannot be used to write an infinite loop (that one leaves by using the `break` statement). The C or C++ idiom is

```
for (;;) {
    // each time through the loop decide if we are ready
    // to leave using a break statement
}
```

The R equivalent is

```
repeat {
    # each time through the loop decide if we are ready
    # to leave using a break expression
}
```

9.4 Comment

> Tradition among experienced S programmers has always been that loops (typically `for` loops) are intrinsically inefficient: expressing computations without loops has provided a measure of entry into the inner circle of S programming.

> John Chambers (*Programming With Data,* p. 173) quoted by the R function `fortune` in the CRAN package `fortunes`}

Sometimes you need loops. Often you don't. We didn't feel the need for any actual examples in this document. We will avoid them whenever we can in this course. Knowledgeable R users know that loops should be avoided when possible, whether or not they have the knowledge to actually avoid loops in any particular case.

10 An Extended Example (Winsorized Means)

10.1 The Problem

Write a function that does Winsorized means. This is defined as replace x of the data from each end with the closest data value that is not removed and then average.

10.2 The Design

That is not enough of a problem statement. We need a more careful statement before we start coding. What values of x are allowed? Obviously we cannot remove more than all of the data so we must have $0 < x \leq 0.5$.

But we can't remove all of the data either, because otherwise "the closest data value that is not removed" makes no sense. So we must have $x < 0.5$.

If there are an odd number of data points, then in order to keep at least one data point, we must have $x < (n-1)/(2n)$. If there are an even number of data points, then in order to keep at least one data point, in which case we must actually keep two because of symmetry, we must have $x < (n-2)/(2n)$.

What if there are zero data points? How should we define the mean of zero data points? What does R do in this case?

```r
mean(double())
```

```
## [1] NaN
```

Presumably we should do the same thing for this case.

What if there are missing data or "not a number" data?

```r
mean(NA)
```

```
## [1] NA
```

```r
mean(NaN)
```

```
## [1] NaN
```

Presumably we should do the same thing for these cases.

Thus our function should look something like this

```r
winsorizedMean <- function(x, winsorize = 0.2) {
    stopifnot(is.numeric(x))
    stopifnot(is.numeric(winsorize))
    stopifnot(length(winsorize) == 1)
    stopifnot(0 < winsorize & winsorize < 0.5)
    n <- length(x)
    if (n == 0) return(NaN)
    if (anyNA(x)) return(NaN)
    # now calculate the Winsorized mean of x in the case where
    # we know length(x) > 0 and there are no NA or NaN in x
}
```

That takes care of the GIEMO issue.

It does not agree with our previous design decision to return `NA` if there are `NA` in the input. This is because

```
anyNA(NA)
```

```
## [1] TRUE
```

```
anyNA(NaN)
```

```
## [1] TRUE
```

does not distinguish between `NA` and `NaN`.

We had never heard of the function `anyNA` before writing this example. We found it discussed on the help page for `is.na`.

Before we can adjust the data, we need it in sorted order because x at each end implicitly refers to the data in sorted order (as laid out on the number line). There is an R function `sort` that does that. Looking at the help page for that, we see that there is a faster function `sort.int` that does partial sorting to make it even faster. Partial sorting seems to be just what we need here. `?sort` says

> If `partial` is not `NULL`, it is taken to contain indices of elements of the result which are to be placed in their correct positions in the sorted array by partial sorting. For each of the result values in a specified position, any values smaller than that one are guaranteed to have a smaller index in the sorted array and any values which are greater are guaranteed to have a bigger index in the sorted array.

If we do a partial sort that gets the two components that we use to replace other components correctly, then we can calculate the Winsorized mean.

So what are those indices? We are supposed to trim `n * winsorize` from each end. But that may not be a round number. We need to round that down to find our how many data values to Winsorize at each end. And how do we round? The help page `?round` describes many functions that do rounding. Reading their descriptions tells us that `floor` rounds down.

So if we define

```
k <- floor(n * winsorize)
```

then the check

```
k < n / 2
```

will do the job of making sure that we have at least one non-replaced data value. Then we want to do a partial sort getting the indices `k + 1` and `n - k` correct.

And then what do we do? We could do the replacement, but we could also just do the arithmetic without doing the replacement

```
(k + 1) * (x[k + 1] + x[n - k]) + sum(x[seq(k + 2, n - k - 1)])
```

divided by `n` is the Winsorized mean, except when `k + 2` $>$ `n - k - 1`, in which case we should be taking the sum of an empty set of numbers (that is, zero) but the R function `seq` produces the sequence that goes *down* from `k + 2` to `n - k - 1` in this case, so that won't work. So we have to special case this.

10.3 The Implementation

This leaves us with

```
winsorizedMean <- function(x, winsorize = 0.2) {
    stopifnot(is.numeric(x))
    stopifnot(is.numeric(winsorize))
    stopifnot(length(winsorize) == 1)
    stopifnot(0 < winsorize & winsorize < 0.5)
    n <- length(x)
    if (n == 0) return(NaN)
    if (anyNA(x)) return(NaN)

    k <- floor(n * winsorize)
    if (k >= n / 2) stop("winsorize too large")
    if (k == 0) return(mean(x))
    x <- sort.int(x, partial = c(k + 1, n - k))
    if (k + 2 >= n - k - 1)
        return(mean(x[c(k + 1, n - k)]))
    else
        return ((k + 1) * (x[k + 1] + x[n - k]) +
            sum(x[seq(k + 2, n - k - 1)])) / n
}
```

10.4 Testing

Once we have an implementation, we are about half done. Now we need to test it to see that it works *correctly*. And if we were putting this in an R package we were writing, we would also have to write a good help page.

The tests should test all the behavior, including that it gives the right errors under the right conditions.

```
winsorizedMean("fred")
```

```
## Error in winsorizedMean("fred"): is.numeric(x) is not TRUE
```

```
winsorizedMean(double())
```

```
## [1] NaN
```

```
winsorizedMean(NA)
```

```
## Error in winsorizedMean(NA): is.numeric(x) is not TRUE
```

```
winsorizedMean(NaN)
```

```
## [1] NaN
```

```
winsorizedMean(1:10, 0.6)
```

```
## Error in winsorizedMean(1:10, 0.6): 0 < winsorize & winsorize < 0.5 is not TRUE
```

```
winsorizedMean(1:10, "fred")
```

```
## Error in winsorizedMean(1:10, "fred"): is.numeric(winsorize) is not TRUE
```

```
winsorizedMean(1:10, 1:10)
```

```
## Error in winsorizedMean(1:10, 1:10): length(winsorize) == 1 is not TRUE
```

```
winsorizedMean(1:10, -3)
```

```
## Error in winsorizedMean(1:10, -3): 0 < winsorize & winsorize < 0.5 is not TRUE
```

OK, except we did not get the result expected when the input was `NA`. A look at the help page for `NA` tells us that the trouble is that there are NA values of each atomic vector type except `raw` and the default type is `logical`. So try again.

```r
winsorizedMean(NA_real_)
```

```
## [1] NaN
```

```r
winsorizedMean(c(1.1, 2.2, NA))
```

```
## [1] NaN
```

OK.

Now how do we get the "winsorize too large" error message? We actually cannot get that (I think). If `winsorize` is less than 1/2, then `k` should be less than `n / 2`.

I confess that when I was writing this code, I forgot to put in checks that the `winsorize` argument was OK when I first wrote this function. Without those checks, the check about `k >= n / 2` does do something important. While I was writing these tests, I realized I should be checking whether both arguments were wrong, and went back and inserted checks for the `winsorize` argument. So now the check about `k >= n / 2` does nothing as far as I can see, but I left it in the code, because it does check a condition that we need to be true in order for the function to work correctly. So if our analysis that it can never be triggered is wrong, it is there to save us.

Now we need to test each code path that calculates a non-error result. There are three of them.

```r
# make up some data
x <- rnorm(20)
identical(winsorizedMean(x, 0.001), mean(x))
```

```
## [1] TRUE
```

So the path where the Winsorized mean is just the mean seems to work.

When do we have `k + 2 >= n - k - 1`? We have equality when

$$2k = n - 3$$

and this means n must be odd. With our even-length data, we have $(n - 3)/2 = 17/2 = 8.5$ and k is greater than that when $k = 9$. And that should happen when `winsorize` is greater than 9/20.

```r
identical(winsorizedMean(x, 0.499), median(x))
```

```
## [1] TRUE
```

The reason why this test works is that in both cases the function is returning the average of the two middle values.

Now for the last case, which does something really complicated and tricky. How do we test that? My answer, developed over long experience in programming in R and C and many other languages, is that we redo the calculation in the least tricky, most straightforward way possible, a way that is hopefully obviously correct. If we get the same answers both ways, we appear to be good.

```r
w1 <- winsorizedMean(x)
sx <- sort(x)
ilow <- seq_along(x) < length(x) * 0.2
ihig <- seq_along(x) > length(x) * 0.8
x.nonrep <- sx[! (ilow & ihig)]
sx[ilow] <- min(x.nonrep)
sx[ihig] <- max(x.nonrep)
w2 <- mean(sx)
identical(w1, w2)
```

```
## [1] FALSE
```

What happened?

```
w1 - w2
```

```
## [1] -2.174608
```

It looks like our function is broken. But it may be that the test is wrong. So first we test the test.

```
sort(x)
```

```
##  [1] -3.49290641 -2.95357980 -2.83756309 -1.34087504 -1.00310455 -0.78182847
##  [7] -0.50945013 -0.31734752 -0.28206969 -0.11876296  0.03776714  0.16022185
## [13]  0.30327340  0.37355389  0.58092897  0.62265967  0.62524329  0.98139727
## [19]  1.49877604  1.78178702
```

```
sx
```

```
##  [1] -3.49290641 -3.49290641 -3.49290641 -1.34087504 -1.00310455 -0.78182847
##  [7] -0.50945013 -0.31734752 -0.28206969 -0.11876296  0.03776714  0.16022185
## [13]  0.30327340  0.37355389  0.58092897  0.62265967  1.78178702  1.78178702
## [19]  1.78178702  1.78178702
```

Our “hopefully obviously correct” method was wrong. Here we have $n * \text{winsorize} = 20 \times 0.2 = 4$, so we should replace the four lowest values with the fifth lowest. And we certainly did not do that.

But first let us check that we are not making another mistake.

```
floor(length(x) * 0.2)
```

```
## [1] 4
```

OK. At least our analysis that we should be replacing the four lowest and four highest is correct (despite inexactness of computer arithmetic).

```
ilow
```

```
##  [1]  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [13] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

```
ihig
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [13] FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE
```

Clearly, we wanted `<=` rather than `<` in our definitions of these.

```
ilow <- seq_along(x) <= length(x) * 0.2
ihig <- seq_along(x) >= length(x) * 0.8
ilow
```

```
##  [1]  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [13] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

```
ihig
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [13] FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
```

If you so the arithmetic counting on your fingers, $20 \times 0.8 = 16$, so R is right, and my notion that this should do the Right Thing is Wrong. Try again.

```
ihig <- rev(ilow)
ihig
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [13] FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE
```

So now those are right.

```
sx <- sort(x)
sx
```

```
##  [1] -3.49290641 -2.95357980 -2.83756309 -1.34087504 -1.00310455 -0.78182847
##  [7] -0.50945013 -0.31734752 -0.28206969 -0.11876296  0.03776714  0.16022185
## [13]  0.30327340  0.37355389  0.58092897  0.62265967  0.62524329  0.98139727
## [19]  1.49877604  1.78178702
```

```
x.nonrep <- sx[! (ilow & ihig)]
x.nonrep
```

```
##  [1] -3.49290641 -2.95357980 -2.83756309 -1.34087504 -1.00310455 -0.78182847
##  [7] -0.50945013 -0.31734752 -0.28206969 -0.11876296  0.03776714  0.16022185
## [13]  0.30327340  0.37355389  0.58092897  0.62265967  0.62524329  0.98139727
## [19]  1.49877604  1.78178702
```

Oops! That was just dumb. I guess I needed a logical or rather than logical and or maybe just the parentheses in different places.

```
x.nonrep <- sx[(! ilow) & (! ihig)]
x.nonrep
```

```
##  [1] -1.00310455 -0.78182847 -0.50945013 -0.31734752 -0.28206969 -0.11876296
##  [7]  0.03776714  0.16022185  0.30327340  0.37355389  0.58092897  0.62265967
```

OK. Now for the rest of the check.

```
sx[ilow] <- min(x.nonrep)
sx[ihig] <- max(x.nonrep)
w2 <- mean(sx)
identical(w1, w2)
```

```
## [1] FALSE
```

What happened?

```
w1 - w2
```

```
## [1] -2.333141
```

so it is not just inexactness of computer arithmetic.

```
sort(x)
```

```
##  [1] -3.49290641 -2.95357980 -2.83756309 -1.34087504 -1.00310455 -0.78182847
##  [7] -0.50945013 -0.31734752 -0.28206969 -0.11876296  0.03776714  0.16022185
## [13]  0.30327340  0.37355389  0.58092897  0.62265967  0.62524329  0.98139727
## [19]  1.49877604  1.78178702
```

```
sx
```

```
##  [1] -1.00310455 -1.00310455 -1.00310455 -1.00310455 -1.00310455 -0.78182847
##  [7] -0.50945013 -0.31734752 -0.28206969 -0.11876296  0.03776714  0.16022185
## [13]  0.30327340  0.37355389  0.58092897  0.62265967  0.62265967  0.62265967
## [19]  0.62265967  0.62265967
```

That sure looks like what we are supposed to average to calculate the Winsorized mean. So now that we think the check is right, it appears that our function is buggy.

So now we have to debug it.

```
n <- length(x)
winsorize <- 0.2
k <- floor(n * winsorize)
k
```

```
## [1] 4
```

```
myx <- sort.int(x, partial = c(k + 1, n - k))
myx
```

```
##  [1] -3.49290641 -2.83756309 -2.95357980 -1.34087504 -1.00310455 -0.50945013
##  [7] -0.78182847  0.16022185 -0.11876296 -0.28206969 -0.31734752  0.03776714
## [13]  0.30327340  0.37355389  0.58092897  0.62265967  0.62524329  0.98139727
## [19]  1.49877604  1.78178702
```

```
w3 <- (k + 1) * (myx[k + 1] + x[n - k]) +
            sum(myx[seq(k + 2, n - k - 1)]) / n
identical(w1, w3)
```

```
## [1] FALSE
```

Oh! In copying this code from the function definition, I found an error. A parenthesis was out of place.

10.5 Re-Implementation

```
winsorizedMean <- function(x, winsorize = 0.2) {
    stopifnot(is.numeric(x))
    stopifnot(is.numeric(winsorize))
    stopifnot(length(winsorize) == 1)
    stopifnot(0 < winsorize & winsorize < 0.5)
    n <- length(x)
    if (n == 0) return(NaN)
    if (anyNA(x)) return(NaN)

    k <- floor(n * winsorize)
    if (k >= n / 2) stop("winsorize too large")
    if (k == 0) return(mean(x))
    x <- sort.int(x, partial = c(k + 1, n - k))
    if (k + 2 >= n - k - 1)
        return(mean(x[c(k + 1, n - k)]))
    else
        return(((k + 1) * (x[k + 1] + x[n - k]) +
            sum(x[seq(k + 2, n - k - 1)])) / n)
}
```

Actually, I realized I had more than one parenthesis mistake in the last expression. Hopefully, this works.

10.6 Re-Test

```
w1 <- winsorizedMean(x)
identical(w1, w2)
```

```
## [1] FALSE
```

```
w1 - w2
```

```
## [1] -1.387779e-17
```

```r
all.equal(w1, w2)
```

```
## [1] TRUE
```

We should have remembered the slogan

> Never test objects of type `"double"` for equality! Use the R function `all.equal` instead (which tests for close to equal, using an optional argument `tolerance` to say how close).

So our function works.

And we also have to do all the rest of the tests.

```r
winsorizedMean("fred")
```

```
## Error in winsorizedMean("fred"): is.numeric(x) is not TRUE
```

```r
winsorizedMean(double())
```

```
## [1] NaN
```

```r
winsorizedMean(NA_real_)
```

```
## [1] NaN
```

```r
winsorizedMean(NaN)
```

```
## [1] NaN
```

```r
winsorizedMean(1:10, 0.6)
```

```
## Error in winsorizedMean(1:10, 0.6): 0 < winsorize & winsorize < 0.5 is not TRUE
```

```r
winsorizedMean(1:10, "fred")
```

```
## Error in winsorizedMean(1:10, "fred"): is.numeric(winsorize) is not TRUE
```

```r
winsorizedMean(1:10, 1:10)
```

```
## Error in winsorizedMean(1:10, 1:10): length(winsorize) == 1 is not TRUE
```

```r
winsorizedMean(1:10, -3)
```

```
## Error in winsorizedMean(1:10, -3): 0 < winsorize & winsorize < 0.5 is not TRUE
```

```r
identical(winsorizedMean(x, 0.001), mean(x))
```

```
## [1] TRUE
```

```r
identical(winsorizedMean(x, 0.499), median(x))
```

```
## [1] TRUE
```

I was happy with this for a day or two, but then I started to wonder about the definition of the sample median being different for even and odd sample sizes. Is this still OK for odd n?

```r
x <- rnorm(21)
winsorizedMean("fred")
```

```
## Error in winsorizedMean("fred"): is.numeric(x) is not TRUE
```

```r
winsorizedMean(double())
```

```
## [1] NaN
```

```
winsorizedMean(NA_real_)
```

```
## [1] NaN
```

```
winsorizedMean(NaN)
```

```
## [1] NaN
```

```
winsorizedMean(1:10, 0.6)
```

```
## Error in winsorizedMean(1:10, 0.6): 0 < winsorize & winsorize < 0.5 is not TRUE
```

```
winsorizedMean(1:10, "fred")
```

```
## Error in winsorizedMean(1:10, "fred"): is.numeric(winsorize) is not TRUE
```

```
winsorizedMean(1:10, 1:10)
```

```
## Error in winsorizedMean(1:10, 1:10): length(winsorize) == 1 is not TRUE
```

```
winsorizedMean(1:10, -3)
```

```
## Error in winsorizedMean(1:10, -3): 0 < winsorize & winsorize < 0.5 is not TRUE
```

```
identical(winsorizedMean(x, 0.001), mean(x))
```

```
## [1] TRUE
```

```
identical(winsorizedMean(x, 0.499), median(x))
```

```
## [1] TRUE
```

```
w1 <- winsorizedMean(x)
sx <- sort(x)
ilow <- seq_along(x) <= length(x) * 0.2
ihig <- rev(ilow)
x.nonrep <- sx[(! ilow) & (! ihig)]
sx[ilow] <- min(x.nonrep)
sx[ihig] <- max(x.nonrep)
w2 <- mean(sx)
identical(w1, w2)
```

```
## [1] TRUE
```

11 Summary

1. R good!
2. Functions good!
3. Vectors good!
4. Vectorizing functions and operators good!
5. Higher-order functions good!
6. Global variables bad, except when they aren't.
7. Garbage in, error messages out (GIEMO)!
8. Design before you code!
9. Test to assure correctness!
10. Keep user interfaces simple. Don't make users memorize crazy special cases like the R function `sample` does. KISS (keep it simple, stupid)!