Chapter 13

Introduction to Numerical Methods in R

13.1 Introduction

This chapter begins with a review of some concepts that should be understood by any statistician who will apply numerical methods that are implemented in statistical packages such as R. Following this introduction, a selection of examples are presented that illustrate the application of numerical methods using functions provided in R. Readers should refer to one or more of the relevant references for a thorough and rigorous presentation of the underlying principles.

Many excellent references are available on numerical methods. Two recent texts written to address the problems of statistical computing in particular are Monahan [210] and Lange [176]. The Monahan text is an excellent resource for statisticians with a limited background in numerical analysis. Nocedal and Wright [217] is a graduate level text on optimization. Lange [175] is another graduate level optimization text that features statistical applications. Cortez [59] covers optimization with algorithms implemented in R. Thisted [284] covers numerical computation for statistics, including numerical analysis, numerical integration, and smoothing. Givens and Hoeting [129] include several chapters on optimization, EM algorithm, and numerical integration.

Computer representation of real numbers

A positive decimal number x is represented by the ordered coefficients $\{d_j\}$ in the series

$$d_n 10^n + d_{n-1}10^{n-1} + \cdots + d_1 10^1 + d_0 + d_{-1}10^{-1} + d_{-2}10^{-2} + \ldots$$

and decimal point separating d_0 and d_{-1}, where d_j are integers in $\{0, 1, \ldots, 9\}$. The same number can be represented in base 2 using the binary digits $\{0, 1\}$ by $a_k a_{k-1} \ldots a_1 a_0 . a_{-1} a_{-2} \ldots$, where

$$x = a_k 2^k + a_{k-1}2^{k-1} + \cdots + a_1 2 + a_0 + a_{-1}2^{-1} + a_{-2}2^{-2} + \ldots,$$

$a_j \in \{0, 1\}$. The point separating a_0 and a_{-1} is called the radix point. Similarly, x can be represented in any integer base $b > 1$ by expanding in powers of b.

R Note 13.1

The function `digitsBase` in the package `sfsmisc` [188] returns the vector of digits that represent a given integer in another base.

Whenever a computer is involved in mathematical calculations, it is very likely to involve the conversions "from" and "to" decimal, because machines and humans represent numbers in different bases. Both types of conversions introduce errors that could be significant in certain cases.

At the lowest level, the computer recognizes exactly two states, like a switch that is on or off, or a circuit that is open or closed. Therefore, at some level, the base 2 representation is used in computer arithmetic. Other powers of 2 such as 8 (octal) or 16 (hexadecimal) are also more natural for low-level routines than base 10.

Positive integers can always be represented by a finite sequence of digits, ending with an implicit radix point. For this reason, integers are called *fixed point* numbers. Numbers that require an explicit radix point in the sequence of digits may be fixed point or floating point (generally treated as floating point in calculations). *Floating point* numbers are represented by a sign, a finite sequence of digits, and an exponent, similar to the representation of real numbers in scientific notation. In general, this representation of a real number is approximate, not exact.

Even though the internal representation of numbers is usually transparent to the user, who conveniently interacts with the software in the decimal system, it is important in statistical computing to understand that there are fundamental differences between mathematical calculations and computer calculations. Mathematical ideas such as *limit*, *supremum*, *infimum*, etc. cannot be exactly reproduced in the computer. No computer has infinite storage capacity, so only finitely many numbers can be represented in the computer; there is a smallest and a largest positive number. See Monahan [210, Ch. 2] for a discussion of fixed point and floating point arithmetic, and inaccuracies that can occur in algorithms as simple as calculation of sample variance.

R Note 13.2

The R variable `.Machine` holds machine specific constants with information on the largest integer, smallest number, etc. For example, in R-2.5.0 for Windows, the largest integer (`.Machine$integer.max`) is $2^{31} - 1 = 2147483647$. Type `.Machine` at the command prompt for the complete list. For portability and reusability of code, tolerances or convergence criteria should be given in terms of machine constants. For example, the `uniroot` function, which seeks a root of a univariate function, has a default tolerance of `.Machine$double.eps^0.25`.

Occasionally users are surprised to find that some mathematical identities appear to be contradicted by the software. A typical example is

```
> (.3 - .1)
[1] 0.2
> (.3 - .1) == .2
[1] FALSE
> .2 - (.3 - .1)
[1] 2.775558e-17
```

The base 2 representation of 0.2 is an infinite series of digits 0.00110011 . . . , which cannot be represented exactly in the computer. Notice that although the result above is not exactly equal to 0.2, the error is negligible. Good programming practice avoids testing the equality of two floating point numbers.

Example 13.1 (Identical and nearly equal)**.** R provides the function `all.equal` to check for near equality of two R objects. In a logical expression, use `isTRUE` to obtain a logical value.

```
> isTRUE(all.equal(.2, .3 - .1))
[1] TRUE
> all.equal(.2, .3)          #not a logical value
[1] "Mean relative  difference: 0.5"
> isTRUE(all.equal(.2, .3))  #always a logical value
[1] FALSE
```

The `isTRUE` function is applied in Example 13.9. The `identical` function is available for testing whether two objects are identical. The help topic for `identical` gives very clear and explicit advice to programmers: "A call to `identical` is the way to test exact equality in `if` and `while` statements, as well as in logical expressions that use `&&` or `||`. In all these applications you need to be assured of getting a single logical value." Also see the examples below.

```
> x <- 1:4
> y <- 2
```

```
> y == 2
[1] TRUE
> x == y  #not necessarily a single logical value
[1] FALSE  TRUE FALSE FALSE
> identical(x, y)  #always a single logical value
[1] FALSE
> identical(y, 2)
[1] TRUE
```

◇

Overflow occurs when the result of an arithmetic operation exceeds the maximum floating point number that can be represented. Underflow occurs when the result is smaller than the minimum floating point number. In the case of underflow, the result might unexpectedly be returned as zero. This could lead to division by zero or other problems that produce unexpected and possibly inaccurate results – without warning. Overflow is usually more obvious, but should be avoided. Good algorithms should set underflows to zero and give a warning if this may produce unexpected results. Programmers can avoid many of these problems, however, by carefully coding arithmetic expressions with the limitations of the machine in mind.

Often the expression to be evaluated is not impossible to compute, but one needs to be careful about the order of operations. One of the most common, and easily avoided problems occurs when we need to compute a ratio of two very large or very small numbers. For example, $n!/(n-2)! = n(n-1)$, but we could easily have trouble computing the numerator or denominator if n is large. A good approach for this type of problem is to take the logarithm of the quotient and exponentiate the result. A typical example that arises in statistical applications is the following.

Example 13.2 (Ratio of two large numbers)**.** Evaluate

$$\frac{\Gamma((n-1)/2)}{\Gamma(1/2)\Gamma((n-2)/2)}.$$

This could be coded using the `gamma` function in R, but $\Gamma(n) = (n-1)!$, so when n is large, `gamma` may return `Inf` and the arithmetic operations could return `NaN`. On the other hand, although numerator and denominator are both large, the ratio is much smaller. Compute the ratio $\Gamma((n-1)/2)/\Gamma((n-2)/2)$ using the logarithm of the gamma function `lgamma`. That is, $\Gamma(n)/\Gamma(m) =$ `exp(lgamma(n) - lgamma(m))`. Also, recall that $\Gamma(1/2) = \sqrt{\pi}$.

```
> n <- 400
> (gamma((n-1)/2) / (sqrt(pi) * gamma((n-2)/2)))
[1] NaN
> exp(lgamma((n-1)/2) - lgamma((n-2)/2)) / sqrt(pi)
[1] 7.953876
```

Chapter 15

Programming Topics

15.1 Introduction

This chapter covers selected programming topics for more experienced R users. The selection of topics is by no means comprehensive, but rather a collection of information that tends to be helpful for developing computational programs, such as benchmarking code or finding the source code of functions. Interested readers can find many published and online resources; some good general references are "Advanced R" [312] (online at `http://adv-r.had.co.nz/`) and "Efficient R Programming" [127] (online at `https://csgillespie.github.io/efficientR/`).

Packages used in the examples of this chapter are microbenchmark, rbenchmark, MASS, mvtnorm, profvis, pryr, and ggplot2. The Rcpp package [82] is used in Section 15.6. In addition, the dplyr, knitr, and Lahman packages are used in the examples of Section 15.7.

15.2 Benchmarking: Comparing the Execution Time of Code

To improve the execution time of R code, it is helpful to compare the timings of different expressions that are equivalent ways to perform a computing task. Often a small chunk of code is responsible for a large chunk of computing time, particularly if the code is executed repeatedly within a program. Users may sometimes be surprised to see that some familiar methods are slower than other convenient alternative methods. This chapter includes several examples and exercises that show how to investigate the performance of code in terms of computing time.

We will use two external packages for benchmarking in this chapter, *microbenchmark* and *rbenchmark*. These packages can be installed from CRAN using the Package Install dialog in RStudio or by entering `install.packages(c("rbenchmark", "microbenchmark"))` in the Console. Other R packages are available for benchmarking; however, for most applica-

tions that would likely correspond to the topics in this book, *microbenchmark* is recommended.

Benchmarks are of course most useful for relative timing comparisons. Absolute timing can vary greatly, depending on the speed of the processor, memory available, background tasks running, and other issues. To compensate for the variance due to background tasks running (such as anti-virus, backup, indexing, cloud services, etc.), benchmark utilities repeat the timing several times (say, 100 times) and report either the average or a summary of the distribution of times including the average.

As we are most interested in the relative performance of some operation compared to others, a statistic such as the ratio of computing times is helpful. A graphical display of the distribution of times is very helpful.

15.2.1 Using the *microbenchmark* Package

The *microbenchmark* package [203] provides a convenient utility function `microbenchmark` that performs sub-millisecond accurate timing of one or more expressions. The documentation for `microbenchmark` states: "This function is only meant for micro-benchmarking small pieces of source code and to compare their relative performance characteristics. You should generally avoid benchmarking larger chunks of your code using this function. Instead, try using the R profiler to detect hot spots and consider rewriting them in C/C++ or FORTRAN." For longer running pieces of code another option is provided in the *rbenchmark* package, discussed in Section 15.2.2.

The `microbenchmark` function displays its results in a summary table, which has the same information as the R `summary` function. The results can be displayed graphically, which is much easier to interpret. Parallel boxplots or the *ggplot2* violin plot can be displayed. The violin plots combine a density estimate for each set of times in a display similar to parallel boxplots. In violin plots, the density is reflected along the horizontal axis, which gives it the "violin" shape. For the timings, the violin plot uses a logarithmic scale which helps compare times that may be of different orders of magnitude.

Example 15.1 (Benchmarking methods to generate a sequence)**.** Several different methods for creating a sequence of integers are available in R. Interestingly, the timings obtained with the `microbenchmark` function reveal that these functions are not equally efficient.

In this example, we use the `microbenchmark` function to compare timings. We also load the *ggplot2* package for a nice graphical display of the results. Saving the return value allows us to display both the summary and the plot of the results.

Three methods of generating the sequence $\{1, 2, 3, \ldots, 1000\}$ will be timed. First, let us check whether the three methods have identical results. The `str` (structure) function is convenient for this comparison.

```
s1 <- 1:10
s2 <- seq(1, 10, 1)
s3 <- seq.int(1, 10, 1)
df <- data.frame(s1=s1, s2=s2, s3=s3)
str(df)

'data.frame':        10 obs. of  3 variables:
 $ s1: int  1 2 3 4 5 6 7 8 9 10
 $ s2: num  1 2 3 4 5 6 7 8 9 10
 $ s3: num  1 2 3 4 5 6 7 8 9 10
```

Although these sequences have equal values, they are stored as different types. Only the first one is stored as type integer. Below we will see that the first approach is also the fastest of the three.

```
library(microbenchmark)
library(ggplot2)

n <- 1000
mb <- microbenchmark(
  seq(1, n, 1),
  seq.int(1, n, 1),
  1:n,
)

mb
autoplot(mb)  # display a violin plot
```

This produces the summary:

```
Unit: nanoseconds
             expr   min    lq     mean median    uq    max neval
     seq(1, n, 1) 32000 35358 38483.09  35950 37136 107852   100
 seq.int(1, n, 1)  1185  1975  2228.20   1976  2370   7506   100
              1:n   395   395   849.41    790   790   3951   100
```

and the violin plot (from *ggplot2*) shown in Figure 15.1. Optionally, `boxplot(mb)` displays the results in parallel boxplots.

Each expression was evaluated 100 times (neval) by default, and the data summary for each set of timings is reported in the microbenchmark summary table. The unit for timing and the number of times to evaluate expressions can be changed.

A nanosecond is one-billionth of a second, or 10^{-9} seconds (`unit = "ns"`). For other problems one may prefer to specify microseconds (`unit = "us"`) or milliseconds (`unit = "ms"`), which are millionth of seconds or thousandth of seconds, respectively.

The summary shows that `1:n` is the fastest way to generate this sequence, while `seq(1, n, 1)` is the slowest. Note, however, that `seq` and `seq.int` may have different types of return values. If the sequence is an integer sequence,

clearly `seq.int` is much more efficient than `seq` (about 17 times faster in this example). ◇

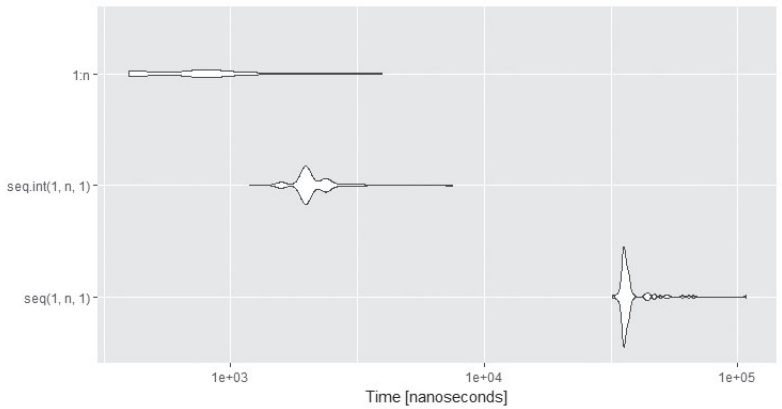


FIGURE 15.1: Violin plot of microbenchmark in Example 15.1 comparing three methods to create a sequence of 1000 integers.

Example 15.2 (Benchmarking methods to initialize a vector)**.** There are several ways to initialize storage for a numeric vector size n to the value 1. Occasionally, we need the equivalent of MATLAB's 'ones' which is a one-column matrix rather than a vector in R. Readers should verify that each of the methods below produces the desired result, either as a numeric vector of ones of length n, or as a $n \times 1$ numeric matrix of ones.

Below, in the arguments to `microbenchmark`, some optional labels are inserted to the left of each expression, to simplify the labeling on the result.

```
n <- 100
mb2 <- microbenchmark(
  numeric = numeric(n) + 1,
  rep = rep(1, n),
  seq = seq(from=1, to=1, length=n),
  ones = matrix(1, nrow=n, ncol=1),
  as.ones = as.matrix(rep(1, n))
)
```

The summary table is below.

```
mb2
Unit: nanoseconds
    expr  min   lq    mean median   uq   max neval
 numeric  395  790  936.35    790  790 10667   100
     rep    0  395  422.76    395  395  3160   100
     seq 5926 6321 7443.04   6716 7111 37136   100
```

```
      ones  790 1185 1438.08   1185 1186 17383   100
   as.ones 4740 5135 6052.37   5136 5531 64395   100
```

This benchmark result shows that if a numeric vector of ones is required, `rep` is the fastest in this comparison. If a numeric matrix (like MATLAB's 'ones') is required, directly initializing the matrix (labeled 'ones') is about five times faster than converting the vector to a matrix (labeled 'as.ones').

An interesting observation is that apparently `as.matrix` is a somewhat costly operation. The average time for evaluating the 'as.ones' expression, which applies `as.matrix` after `rep`, is more than three times the sum of the average times for 'rep' and 'ones'.

◇

R Note 15.1 The 'ones' column vector in R

The $n \times 1$ matrix of ones is sometimes required in computing with matrices. In R a numeric vector and a numeric matrix are different types of objects. A vector of ones does not have a dimension attribute, so it cannot be used in matrix multiplication or as an argument to a function that requires it have a dimension. To create a 'ones' matrix in R, use `matrix` to create the object. Notice that the structure (`str`) of the vector and the matrix differ, and the vector does not have a dimension attribute.

```
> onesVector <- rep(1, 5)
> onesMatrix <- matrix(1, nrow=5)
> str(onesVector)
 num [1:5] 1 1 1 1 1
> str(onesMatrix)
 num [1:5, 1] 1 1 1 1 1
> dim(onesVector)
NULL
> dim(onesMatrix)
[1] 5 1
```

15.2.2 Using the *rbenchmark* Package

The *rbenchmark* package [172] provides a function `benchmark`, which is a convenient wrapper around the R `system.time` function. (Note that there is also a package named *benchmark* available, but that is not the general purpose utility that we want here; for this type of example, install the package *rbenchmark* rather than *benchmark*).

The `benchmark` function performs a similar timing experiment as `microbenchmark`. It replicates timing the list of expressions a default number of times (100) and returns a table summarizing the average times and the relative timings, depending on which columns were specified. The individual

replicates of times are not returned, so it is not possible to display these results in a boxplot or violin plot as with `microbenchmark`.

When timing a relatively simple expression with `benchmark`, the time may be reported as zero. In that case, the relative time column will have NA. In this case, it is probably better to use `microbenchmark` for its sub-millisecond accuracy.

Example 15.3 (Timings of two multivariate normal generators)**.** In Example 3.19 we compared the timings of several different functions to generate multivariate normal random samples. In that example, we used the `system.time` function directly. Here we revisit the example using the wrapper for `system.time` provided by *rbenchmark*. In this example, we compare two of the generators, `MASS::mvrnorm` and `mvtnorm::rmvnorm`.

```
library(rbenchmark)
library(MASS)
library(mvtnorm)
n <- 100          #sample size
d <- 30           #dimension
N <- 2000         #iterations
mu <- numeric(d)

benchmark(
    columns = c("test", "replications", "elapsed", "relative"),
    replications = 2000,
    cov = {S <- cov(matrix(rnorm(n*d), n, d))},
    mvrnorm = mvrnorm(n, mu, S),
    rmvnorm = rmvnorm(n, mu, S)
)

     test replications elapsed relative
1     cov         2000    0.76    1.000
2 mvrnorm         2000    1.09    1.434
3 rmvnorm         2000    1.50    1.974
```

This table includes the efficiency of the methods relative to the fastest one. Here we included timing to generate the random covariance matrices, because that is not a negligible amount of time. The relative efficiencies are $1.09/0.76 = 1.434211$ and $1.50/0.76 = 1.973684$. If we were to take `mvrnorm` as the baseline, the relative speed of `rmvnorm` is $1.50/1.09 = 1.377$ or equivalently, $1.974/1.434 = 1.377$. In this benchmark, `MASS::mvrnorm` (using spectral decomposition) is faster than `rmvnorm` by about 27%, because $(1.50 - 1.09)/1.50 \approxeq 0.27$.

Referring to Example 3.19, we have similar results for these two generators in terms of relative performance. ◇

15.3 Profiling

To gain some insight into what parts of some code are less efficient, the basic idea is to perform many repeated micro-timings. Instead of timing an entire function call or block of code, the code is stopped at regular, small increments of time, and information about what is executing at that moment is recorded. See "Tidying and Profiling Code" in the R help file "Writing R Extensions" [230].

Example 15.4 (Profiling with Rprof)**.** We apply `Rprof` here to the distance correlation function `dcor` in the *energy* package, placing the function we want to profile between the `Rprof` calls.

```
x <- rnorm(1000)
y <- rnorm(1000)

Rprof("pr.out", line.profiling = TRUE)
energy::dcor(x, y)
Rprof(NULL)
```

Now we can use `summaryRprof` to summarize the results.

```
> summaryRprof("pr.out")
$`by.self`
                 self.time self.pct total.time total.pct
"as.matrix.dist"      0.08     50.0       0.12      75.0
".C"                  0.02     12.5       0.02      12.5
"col"                 0.02     12.5       0.02      12.5
"matrix"              0.02     12.5       0.02      12.5
"t.default"           0.02     12.5       0.02      12.5

$by.total
                 total.time total.pct self.time self.pct
".dcov"                0.16     100.0      0.00      0.0
"energy::dcor"         0.16     100.0      0.00      0.0
"as.matrix.dist"       0.12      75.0      0.08     50.0
"as.matrix"            0.12      75.0      0.00      0.0
".C"                   0.02      12.5      0.02     12.5
"col"                  0.02      12.5      0.02     12.5
"matrix"               0.02      12.5      0.02     12.5
"t.default"            0.02      12.5      0.02     12.5
"t"                    0.02      12.5      0.00      0.0

$sample.interval
[1] 0.02

$sampling.time
[1] 0.16
```

```
    ...     (several lines omitted)

    .Call(C_Cdist, x, method, attrs, p)
}
<bytecode: 0x0000000017278aa8>
<environment: namespace:stats>
```

To view the C code for `C_Cdist`, we can access the R source code if we know where to look. It is convenient to search using the interface on GitHub. To find the file, open a web browser to the GitHub R source code repository at `https://github.com/wch/r-source`. Enter "Cdist" in the search box at the top labeled "This repository" to search the entire r-source repository. The top two results give us the `.Call` in the R function `dist`, and `Cdist` in "distance.c".

Another way to find the file "distance.c" in the r-source repository is to navigate to it through `src/library/stats`, although this assumes that the location of the file is easily guessed. ◇

15.6 Linking C/C++ Code Using *Rcpp*

The *Rcpp* package provides an elegant interface to extend R code with compiled C or C++ functions. In this section, a few simple examples will be explored that give some idea of how easy it is to use, given some basic knowledge of R and C++.

Why use C++ with R? We have several methods to improve the computational efficiency of R code. After benchmarks and profiling code, sometimes slow chunks remain, and the best way to speed up is to write those parts in compiled code. However small or complex the programming task, Rcpp facilitates combining compiled C++ code with R.

Some of the options that Rcpp makes available are

- Use `cppFunction` to include a C++ function in an R script or R Markdown file.
- Use `sourceCpp` to compile and link a C++ file and use any of its exported functions.
- Create a C++ file of functions, compile and test it within RStudio.
- Build packages within RStudio that include C++ code.

Here we only provide a brief introduction to Rcpp, so interested readers are referred to the references [82, 312], for a more thorough discussion and many examples. See also Dirk Eddelbuettel's Rcpp page at `http:`

//dirk.eddelbuettel.com/code/rcpp.html for code examples, slides, and other resources. The Rcpp Gallery at http://gallery.rcpp.org/ is another good resource.

To get started, one should install the Rcpp package. Some additional tools may need to be installed, depending on the operating system. Windows users need to install Rtools, which can be installed from the same page as Windows for R (go to cran.r-project.org). Mac users should install Xcode from the app store.

Example 15.17 (A first Rcpp experiment)**.** To check that Rcpp and tools are installed and ready to use, open and execute the sample C++ file in RStudio. Under the File menu, select "New File ..." and "C++ file". A file appears with many comments. The file is shown (with comments removed) in Listing 15.1.

Listing 15.1: New C++ file from RStudio File menu

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
NumericVector timesTwo(NumericVector x) {
  return x * 2;
}

/*** R
timesTwo(42)
*/
```

Click "Source" and save the file if it is not already saved, say as 'temp.cpp'. The C++ code is compiled and the optional chunk of R code at the bottom executes. The following should appear in the Console:

```
> Rcpp::sourceCpp('temp.cpp')

> timesTwo(42)
[1] 84
```

The first two lines in the file are required. The line

```
// [[Rcpp::export]]
```

is included before each function that is exported; if it is not there, then the function is compiled but not exported.

The argument type of the function 'timesTwo' and the return type are both NumericVector. This makes it easy to work with arrays without using pointers.

The special environment at the bottom makes it possible to insert a few lines of R code to run.

Now, look in the Environment tab. A new object 'timesTwo' has appeared, which has type 'function'. This is an R function. Rcpp has created an R function matching the C++ function and we have called the R function in the expression 'timesTwo(42)'. This function is available to use as long as it remains in the environment. Try it with other arguments.

```
> timesTwo(0:5)
[1]  0  2  4  6  8 10
```

◇

Use cppFunction within an R script

Rcpp provides the function `cppFunction` which dynamically defines a C++ function within an R script or a chunk of R code within R Markdown.

The function definition (the C++ function) is entered as quoted text in the first argument. Other arguments are optional.

Example 15.18 (cppFunction)**.** In this example, we write a C++ function `vecnorm` to compute the Euclidean norm of a vector $\|x\| = \sqrt{\sum_{j=1}^{d} x_j^2}$. The matching R function `vecnorm` will be added to the R environment and it can then be used like any other R function.

```
library(Rcpp)

set.seed(1)
x <- matrix(rnorm(20), nrow = 5, ncol = 4)

cppFunction('double vecnorm(NumericVector v) {
  // compute the Euclidean norm of vector v
  int d = v.size();
  double s = 0.0;
  for (int i = 0; i < d; i++)
    s += v(i) * v(i);
  return sqrt(s);
}')

print(vecnorm(x[, 1]))
print(apply(x, MARGIN = 2, FUN = "vecnorm"))
```

This is an R source file. After saving it and clicking on 'Source', the result in the console is

```
[1] 1.943693
[1] 1.943693 1.371373 2.998961 1.385724
```

◇

Source C++ functions with sourceCpp

Many functions that one would develop with Rcpp would be used by more than one application. It is a good practice to place these functions in a separate C++ source file, so that there is exactly one version of the code to maintain. We have already seen in Example 15.17 that a file of C++ functions that includes the first lines

```
#include <Rcpp.h>
using namespace Rcpp;
```

can be compiled in RStudio by clicking on 'Source'. It is also possible to source one or more C++ files of this type within R code using the Rcpp function `sourceCpp`.

Example 15.19 (sourceCpp)**.** To illustrate the Rcpp `sourceCpp` function, let us create a small demo function for printing. Although there is only one function in the source file below, several functions can be defined in the same source file.

Listing 15.2: printme.cpp

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
void print_me(IntegerVector x) {
  // several ways to print in Rcpp
  Rprintf("my vector is (%d, %d, %d)\n",
          x[0], x[1], x[2]);
  Rcpp::Rcout << "my vector is " << x << std::endl;
  Rf_PrintValue(x);
}
```

Save this file as "printme.cpp". Then load Rcpp and use the `sourceCpp` function within R code to compile and link this function. The result is that a new R function called `print_me` is created and available to call from R.

```
library(Rcpp)
sourceCpp("printme.cpp")

x <- sample(1:5)
print_me(x)
```

Running the lines above produced

```
> x <- sample(1:5)
> print_me(x)
my vector is (4, 5, 3)
my vector is 4 5 3 1 2
[1] 4 5 3 1 2
```