High performance functions with Rcpp

Sometimes R code just isn't fast enough. You've used profiling to figure out where your bottlenecks are, and you've done everything you can in R, but your code still isn't fast enough. In this chapter you'll learn how to improve performance by rewriting key functions in C++. This magic comes by way of the Rcpp package, a fantastic tool written by Dirk Eddelbuettel and Romain Francois (with key contributions by Doug Bates, John Chambers, and JJ Allaire). Rcpp makes it very simple to connect C++ to R. While it is *possible* to write C or Fortran code for use in R, it will be painful by comparison. Rcpp provides a clean, approachable API that lets you write high-performance code, insulated from R's arcane C API.

Typical bottlenecks that C++ can address include:

- Loops that can't be easily vectorised because subsequent iterations depend on previous ones.
- Recursive functions, or problems which involve calling functions millions of times. The overhead of calling a function in C++ is much lower than that in R.
- Problems that require advanced data structures and algorithms that R doesn't provide. Through the standard template library (STL), C++ has efficient implementations of many important data structures, from ordered maps to double-ended queues.

The aim of this chapter is to discuss only those aspects of C++ and Rcpp that are absolutely necessary to help you eliminate bottlenecks in your code. We won't spend much time on advanced features like object oriented programming or templates because the focus is on writing small, self-contained functions, not big programs. A working knowledge of C++ is helpful, but not essential. Many good tutorials and references are freely available, includinghttp://www.learncpp.com/ and http://www.cplusplus.com/. For more advanced topics, the *Effective C++* series by Scott Meyers is popular choice. You may also enjoy Dirk Eddelbuettel's *Seamless R and C++ integration with Rcpp*, which goes into much greater detail into all aspects of Rcpp.

Outline

- Getting started with C++ teaches you how to write C++ by converting simple R functions to their C++ equivalents. You'll learn how C++ differs from R, and what the key scalar, vector, and matrix classes are called.
- Using sourceCpp shows you how to use sourceCpp() to load a C++ file from disk in the same way you usesource() to load a file of R code.
- Attributes & other classes discusses how to modify attributes from Rcpp, and mentions some of the other important classes.
- Missing values teaches you how to work with R's missing values in C++.
- Rcpp sugar discusses Rcpp "sugar", which allows you to avoid loops in C++ and write code that looks very similar to vectorised R code.
- The STL shows you how to use some of the most important data structures and algorithms from the standard template library, or STL, built-in to C++.

- Case studies shows two real case studies where Rcpp was used to get considerable performance improvements.
- Putting Rcpp in a package teaches you how to add C++ code to a package.
- Learning more concludes the chapter with pointers to more resources to help you learn Rcpp and C++.

Prerequistes

All examples in this chapter need version 0.10.1 or above of the Rcpp package. This version includes cppFunction()and sourceCpp(), which makes it very easy to connect C++ to R. Install the latest version of Rcpp from CRAN withinstall.packages("Rcpp").

You'll also need a working C++ compiler. To get it:

- On Windows, install Rtools.
- On Mac, install Xcode from the app store.
- On Linux, sudo apt-get install r-base-dev or similar.

Getting started with C++

cppFunction() allows you to write C++ functions in R:

```
library(Rcpp)
#>
#> Attaching package: 'Rcpp'
#>
#> The following object is masked from 'package:inline':
#>
#>     registerPlugin
cppFunction('int add(int x, int y, int z) {
  int sum = x + y + z;
  return sum;
}')
# add works like a regular R function
add
#> function (x, y, z)
#> .Primitive(".Call")(<pointer: 0x7fbefa7737c0>, x, y, z)
add(1, 2, 3)
#> [1] 6
```

When you run this code, Rcpp will compile the C++ code and construct an R function that connects to the compiled C++ function. We're going to use this simple interface to learn how to write C++. C++ is a large language, and there's no way to cover it all in just one chapter. Instead, you'll get the basics so that you can start writing useful functions to address bottlenecks in your R code.

The following sections will teach you the basics by translating simple R functions to their C++ equivalents. We'll start simple with a function that has no inputs and a scalar output, and then get progressively more complicated:

- Scalar input and scalar output
- Vector input and scalar output
- Vector input and vector output
- Matrix input and vector output

No inputs, scalar output

Let's start with a very simple function. It has no arguments and always returns the integer 1:

```
one <- function() 1L
```

The equivalent C++ function is:

```
int one() {
  return 1;
}
```

We can compile and use this from R with `cppFunction`

```
cppFunction('int one() {
  return 1;
}')
```

This small function illustrates a number of important differences between R and C++:

- The syntax to create a function looks like the syntax to call a function; you don't use assignment to create functions as you do in R.
- You must declare the type of output the function returns. This function returns an `int` (a scalar integer). The classes for the most common types of R vectors are: `NumericVector`, `IntegerVector`, `CharacterVector`, and`LogicalVector`.
- Scalars and vectors are different. The scalar equivalents of numeric, integer, character, and logical vectors are:`double`, `int`, `String`, and `bool`.
- You must use an explicit `return` statement to return a value from a function.
- Every statement is terminated by a `;`.

Scalar input, scalar output

The next example function implements a scalar version of the sign() function which returns 1 if the input is positive, and -1 if it's negative:

```
signR <- function(x) {
  if (x > 0) {
    1
  } else if (x == 0) {
    0
  } else {
    -1
  }
}


cppFunction('int signC(int x) {
  if (x > 0) {
    return 1;
  } else if (x == 0) {
    return 0;
  } else {
    return -1;
  }
}')
```

In the C++ version:

- We declare the type of each input in the same way we declare the type of the output. While this makes the code a little more verbose, it also makes it very obvious what type of input the function needs.
- The if syntax is identical — while there are some big differences between R and C++, there are also lots of similarities! C++ also has a while statement that works the same way as R's. As in R you can use break to exit the loop, but to skip one iteration you need to use continue instead of next.

Vector input, scalar output

One big difference between R and C++ is that the cost of loops is much lower in C++. For example, we could implement the `sum` function in R using a loop. If you've been programming in R a while, you'll probably have a visceral reaction to this function!

```
sumR <- function(x) {
  total <- 0
  for (i in seq_along(x)) {
    total <- total + x[i]
  }
  total
}
```

In C++, loops have very little overhead, so it's fine to use them. In STL, you'll see alternatives to `for` loops that more clearly express your intent; they're not faster, but they can make your code easier to understand.

```
cppFunction('double sumC(NumericVector x) {
  int n = x.size();
  double total = 0;
  for(int i = 0; i < n; ++i) {
    total += x[i];
  }
  return total;
}')
```

The C++ version is similar, but:

- To find the length of the vector, we use the `.size()` method, which returns an integer. C++ methods are called with `.` (i.e., a full stop).
- The `for` statement has a different syntax: `for(init; check; increment)`. This loop is initialised by creating a new variable called `i` with value 0. Before each iteration we check that `i < n`, and terminate the loop if it's not. After each iteration, we increment the value of `i` by one, using the special prefix operator `++` which increases the value of `i` by 1.
- In C++, vector indices start at 0. I'll say this again because it's so important: **IN C++, VECTOR INDICES START AT 0**! This is a very common source of bugs when converting R functions to C++.
- Use `=` for assignment, not `<-`.
- C++ provides operators that modify in-place: `total += x[i]` is equivalent to `total = total + x[i]`. Similar in-place operators are `-=`, `*=`, and `/=`.

This is a good example of where C++ is much more efficient than R. As shown by the following microbenchmark,sumC() is competitive with the built-in (and highly optimised) sum(), while sumR() is several orders of magnitude slower.

```
x <- runif(1e3)
microbenchmark(
  sum(x),
  sumC(x),
  sumR(x)
)
#> Unit: microseconds
#>     expr    min     lq   mean median     uq     max neval
#>   sum(x)   2.82   2.96   3.39   3.17   3.31    12.3   100
#>  sumC(x)   5.98   6.53   7.88   7.13   7.91    37.6   100
#>  sumR(x) 626.00 663.00 758.79 705.00 748.00 2,400.0   100
```

Vector input, vector output

Next we'll create a function that computes the Euclidean distance between a value and a vector of values:

```
pdistR <- function(x, ys) {
  sqrt((x - ys) ^ 2)
}
```

It's not obvious that we want x to be a scalar from the function definition. We'd need to make that clear in the documentation. That's not a problem in the C++ version because we have to be explicit about types:

```
cppFunction('NumericVector pdistC(double x, NumericVector ys) {
  int n = ys.size();
  NumericVector out(n);

  for(int i = 0; i < n; ++i) {
    out[i] = sqrt(pow(ys[i] - x, 2.0));
  }
  return out;
```

```
}')
```

This function introduces only a few new concepts:

- We create a new numeric vector of length `n` with a constructor: `NumericVector out(n)`. Another useful way of making a vector is to copy an existing one: `NumericVector zs = clone(ys)`.
- C++ uses `pow()`, not `^`, for exponentiation.

Note that because the R version is fully vectorised, it's already going to be fast. On my computer, it takes around 8 ms with a 1 million element `y` vector. The C++ function is twice as fast, ~4 ms, but assuming it took you 10 minutes to write the C++ function, you'd need to run it ~150,000 times to make rewriting worthwhile. The reason why the C++ function is faster is subtle, and relates to memory management. The R version needs to create an intermediate vector the same length as `y` (`x - ys`), and allocating memory is an expensive operation. The C++ function avoids this overhead because it uses an intermediate scalar.

In the sugar section, you'll see how to rewrite this function to take advantage of Rcpp's vectorised operations so that the C++ code is almost as concise as R code.

Matrix input, vector output

Each vector type has a matrix equivalent: `NumericMatrix`, `IntegerMatrix`, `CharacterMatrix`, and `LogicalMatrix`. Using them is straightforward. For example, we could create a function that reproduces `rowSums()`:

```
cppFunction('NumericVector rowSumsC(NumericMatrix x) {
  int nrow = x.nrow(), ncol = x.ncol();
  NumericVector out(nrow);

  for (int i = 0; i < nrow; i++) {
    double total = 0;
    for (int j = 0; j < ncol; j++) {
      total += x(i, j);
    }
    out[i] = total;
  }
  return out;
}')

set.seed(1014)
x <- matrix(sample(100), 10)
```

```
rowSums(x)
#>  [1] 458 558 488 458 536 537 488 491 508 528
rowSumsC(x)
#>  [1] 458 558 488 458 536 537 488 491 508 528
```

The main differences:

- In C++, you subset a matrix with (), not [].
- Use .nrow() and .ncol() *methods* to get the dimensions of a matrix.

Using sourceCpp

So far, we've used inline C++ with cppFunction(). This makes presentation simpler, but for real problems, it's usually easier to use stand-alone C++ files and then source them into R using sourceCpp(). This lets you take advantage of text editor support for C++ files (e.g., syntax highlighting) as well as making it easier to identify the line numbers in compilation errors.

Your stand-alone C++ file should have extension .cpp, and needs to start with:

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

And for each function that you want available within R, you need to prefix it with:

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

Note that the space is mandatory.

If you're familiar with roxygen2, you might wonder how this relates to @export. Rcpp::export controls whether a function is exported from C++ to R; @export controls whether a function is exported from a package and made available to the user.

You can embed R code in special C++ comment blocks. This is really convenient if you want to run some test code:

```
/*** R
# This is R code
*/
```

The R code is run with source(echo = TRUE) so you don't need to explicitly print output.

To compile the C++ code, use sourceCpp("path/to/file.cpp"). This will create the matching R functions and add them to your current session. Note that these functions can not be saved in a .Rdata file and reloaded in a later session; they must be recreated each time you restart R. For example, running sourceCpp() on the following file implements mean in C++ and then compares it to the built-in mean():

```
#include <Rcpp.h>

using namespace Rcpp;


// [[Rcpp::export]]

double meanC(NumericVector x) {

  int n = x.size();

  double total = 0;


  for(int i = 0; i < n; ++i) {

    total += x[i];

  }

  return total / n;

}


/*** R

library(microbenchmark)

x <- runif(1e5)

microbenchmark(

  mean(x),

  meanC(x)

)

*/
```

NB: if you run this code yourself, you'll notice that `meanC()` is much faster than the built-in `mean()`. This is because it trades numerical accuracy for speed.

For the remainder of this chapter C++ code will be presented stand-alone rather than wrapped in a call to`cppFunction`. If you want to try compiling and/or modifying the examples you should paste them into a C++ source file that includes the elements described above.

Exercises

With the basics of C++ in hand, it's now a great time to practice by reading and writing some simple C++ functions. For each of the following functions, read the code and figure out what the corresponding base R function is. You might not understand every part of the code yet, but you should be able to figure out the basics of what the function does.

```
double f1(NumericVector x) {
  int n = x.size();
  double y = 0;

  for(int i = 0; i < n; ++i) {
    y += x[i] / n;
  }
  return y;
}

NumericVector f2(NumericVector x) {
  int n = x.size();
  NumericVector out(n);

  out[0] = x[0];
  for(int i = 1; i < n; ++i) {
    out[i] = out[i - 1] + x[i];
  }
  return out;
}

bool f3(LogicalVector x) {
  int n = x.size();

  for(int i = 0; i < n; ++i) {
    if (x[i]) return true;
  }
  return false;
}

int f4(Function pred, List x) {
```

```
  int n = x.size();

  for(int i = 0; i < n; ++i) {
    LogicalVector res = pred(x[i]);
    if (res[0]) return i + 1;
  }
  return 0;
}

NumericVector f5(NumericVector x, NumericVector y) {
  int n = std::max(x.size(), y.size());
  NumericVector x1 = rep_len(x, n);
  NumericVector y1 = rep_len(y, n);

  NumericVector out(n);

  for (int i = 0; i < n; ++i) {
    out[i] = std::min(x1[i], y1[i]);
  }

  return out;
}
```

To practice your function writing skills, convert the following functions into C++. For now, assume the inputs have no missing values.

1. `all()`
2. `cumprod()`, `cummin()`, `cummax()`.
3. `diff()`. Start by assuming lag 1, and then generalise for lag `n`.
4. `range`.
5. `var`. Read about the approaches you can take on wikipedia. Whenever implementing a numerical algorithm, it's always good to check what is already known about the problem.

Attributes and other classes

You've already seen the basic vector classes (IntegerVector, NumericVector, LogicalVector, CharacterVector) and their scalar (int, double, bool, String) and matrix (IntegerMatrix, NumericMatrix, LogicalMatrix, CharacterMatrix) equivalents.

All R objects have attributes, which can be queried and modified with .attr(). Rcpp also provides .names() as an alias for the name attribute. The following code snippet illustrates these methods. Note the use of ::create(), a *class*method. This allows you to create an R vector from C++ scalar values:

```
#include <Rcpp.h>

using namespace Rcpp;


// [[Rcpp::export]]

NumericVector attribs() {

  NumericVector out = NumericVector::create(1, 2, 3);


  out.names() = CharacterVector::create("a", "b", "c");

  out.attr("my-attr") = "my-value";

  out.attr("class") = "my-class";


  return out;

}
```

For S4 objects, .slot() plays a similar role to .attr().

Lists and data frames

Rcpp also provides classes List and DataFrame, but they are more useful for output than input. This is because lists and data frames can contain arbitrary classes but C++ needs to know their classes in advance. If the list has known structure (e.g., it's an S3 object), you can extract the components and manually convert them to their C++ equivalents with as(). For example, the object created by lm(), the function that fits a linear model, is a list whose components are always of the same type. The following code illustrates how you might extract the mean percentage error (mpe()) of a linear model. This isn't a good example of when to use C++, because it's so easily implemented in R, but it shows how to work with an important S3 class. Note the use of .inherits() and the stop() to check that the object really is a linear model.

```
#include <Rcpp.h>

using namespace Rcpp;
```

```
// [[Rcpp::export]]
double mpe(List mod) {
  if (!mod.inherits("lm")) stop("Input must be a linear model");

  NumericVector resid = as<NumericVector>(mod["residuals"]);
  NumericVector fitted = as<NumericVector>(mod["fitted.values"]);

  int n = resid.size();
  double err = 0;
  for(int i = 0; i < n; ++i) {
    err += resid[i] / (fitted[i] + resid[i]);
  }
  return err / n;
}
mod <- lm(mpg ~ wt, data = mtcars)
mpe(mod)
#> [1] -0.0154
```

Functions

You can put R functions in an object of type `Function`. This makes calling an R function from C++ straightforward. We first define our C++ function:

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

// [[Rcpp::export]]
RObject callWithOne(Function f) {
  return f(1);
}
```

Then call it from R:

```
callWithOne(function(x) x + 1)
#> [1] 2
```

```
callWithOne(paste)
#> [1] "1"
```

What type of object does an R function return? We don't know, so we use the catchall type `RObject`. An alternative is to return a `List`. For example, the following code is a basic implementation of `lapply` in C++:

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

// [[Rcpp::export]]
List lapply1(List input, Function f) {
  int n = input.size();
  List out(n);

  for(int i = 0; i < n; i++) {
    out[i] = f(input[i]);
  }

  return out;
}
```

Calling R functions with positional arguments is obvious:

```
f("y", 1);
```

But to use named arguments, you need a special syntax:

```
f(_["x"] = "y", _["value"] = 1);
```

Other types

There are also classes for many more specialised language objects: `Environment`, `ComplexVector`, `RawVector`,`DottedPair`, `Language`, `Promise`, `Symbol`, `WeakReference`, and so on. These are beyond the scope of this chapter and won't be discussed further.

Missing values

If you're working with missing values, you need to know two things:

- how R's missing values behave in C++'s scalars (e.g., double).
- how to get and set missing values in vectors (e.g., NumericVector).

Scalars

The following code explores what happens when you take one of R's missing values, coerce it into a scalar, and then coerce back to an R vector. Note that this kind of experimentation is a useful way to figure out what any operation does.

```
#include <Rcpp.h>

using namespace Rcpp;


// [[Rcpp::export]]

List scalar_missings() {

  int int_s = NA_INTEGER;

  String chr_s = NA_STRING;

  bool lgl_s = NA_LOGICAL;

  double num_s = NA_REAL;


  return List::create(int_s, chr_s, lgl_s, num_s);

}

str(scalar_missings())

#> List of 4

#>  $ : int NA

#>  $ : chr NA

#>  $ : logi TRUE

#>  $ : num NA
```

With the exception of bool, things look pretty good here: all of the missing values have been preserved. However, as we'll see in the following sections, things are not quite as straightforward as they seem.

Integers

With integers, missing values are stored as the smallest integer. If you don't do anything to them, they'll be preserved. But, since C++ doesn't know that the smallest integer has this special behaviour, if you do anything to it you're likely to get an incorrect value: for example, evalCpp('NA_INTEGER + 1') gives -2147483647.

So if you want to work with missing values in integers, either use a length one IntegerVector or be very careful with your code.

Doubles

With doubles, you may be able to get away with ignoring missing values and working with NaNs (not a number). This is because R's NA is a special type of IEEE 754 floating point number NaN. So any logical expression that involves a NaN (or in C++, NAN) always evaluates as FALSE:

```
evalCpp("NAN == 1")
#> [1] FALSE
evalCpp("NAN < 1")
#> [1] FALSE
evalCpp("NAN > 1")
#> [1] FALSE
evalCpp("NAN == NAN")
#> [1] FALSE
```

But be careful when combining then with boolean values:

```
evalCpp("NAN && TRUE")
#> [1] TRUE
evalCpp("NAN || FALSE")
#> [1] TRUE
```

However, in numeric contexts NaNs will propagate NAs:

```
evalCpp("NAN + 1")
#> [1] NaN
evalCpp("NAN - 1")
#> [1] NaN
evalCpp("NAN / 1")
#> [1] NaN
evalCpp("NAN * 1")
#> [1] NaN
```

Strings

`String` is a scalar string class introduced by Rcpp, so it knows how to deal with missing values.

Boolean

While C++'s bool has two possible values (true or false), a logical vector in R has three (TRUE, FALSE, and NA). If you coerce a length 1 logical vector, make sure it doesn't contain any missing values otherwise they will be converted to TRUE.

Vectors

With vectors, you need to use a missing value specific to the type of vector, NA_REAL, NA_INTEGER, NA_LOGICAL,NA_STRING:

```
#include <Rcpp.h>

using namespace Rcpp;


// [[Rcpp::export]]

List missing_sampler() {

  return List::create(

    NumericVector::create(NA_REAL),

    IntegerVector::create(NA_INTEGER),

    LogicalVector::create(NA_LOGICAL),

    CharacterVector::create(NA_STRING));

}

str(missing_sampler())

#> List of 4

#>  $ : num NA

#>  $ : int NA

#>  $ : logi NA

#>  $ : chr NA
```

To check if a value in a vector is missing, use the class method ::is_na():

```
#include <Rcpp.h>

using namespace Rcpp;


// [[Rcpp::export]]

LogicalVector is_naC(NumericVector x) {

  int n = x.size();

  LogicalVector out(n);
```

```
  for (int i = 0; i < n; ++i) {
    out[i] = NumericVector::is_na(x[i]);
  }
  return out;
}
is_naC(c(NA, 5.4, 3.2, NA))
#> [1]  TRUE FALSE FALSE  TRUE
```

Another alternative is the sugar function is_na(), which takes a vector and returns a logical vector.

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

// [[Rcpp::export]]
LogicalVector is_naC2(NumericVector x) {
  return is_na(x);
}
is_naC2(c(NA, 5.4, 3.2, NA))
#> [1]  TRUE FALSE FALSE  TRUE
```

Exercises

1. Rewrite any of the functions from the first exercise to deal with missing values. If na.rm is true, ignore the missing values. If na.rm is false, return a missing value if the input contains any missing values. Some good functions to practice with are min(), max(), range(), mean(), and var().
2. Rewrite cumsum() and diff() so they can handle missing values. Note that these functions have slightly more complicated behaviour.

Rcpp sugar

Rcpp provides a lot of syntactic “sugar” to ensure that C++ functions work very similarly to their R equivalents. In fact, Rcpp sugar makes it possible to write efficient C++ code that looks almost identical to its R equivalent. If there’s a sugar version of the function you’re interested in, you should use it: it’ll be both expressive and well tested. Sugar functions aren’t always faster than a handwritten equivalent, but they will get faster in the future as more time is spent on optimising Rcpp.

Sugar functions can be roughly broken down into

- arithmetic and logical operators
- logical summary functions
- vector views
- other useful functions

Arithmetic and logical operators

All the basic arithmetic and logical operators are vectorised: `+` *, `-`, /, `pow`, `<`, `<=`, `>`, `>=`, `==`, `!=`, !. For example, we could use sugar to considerably simplify the implementation of `pdistC()`.

```
pdistR <- function(x, ys) {
  sqrt((x - ys) ^ 2)
}
```

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


// [[Rcpp::export]]
NumericVector pdistC2(double x, NumericVector ys) {
  return sqrt(pow((x - ys), 2));
}
```

Logical summary functions

The sugar function `any()` and `all()` are fully lazy so that `any(x == 0)`, for example, might only need to evaluate one element of a vector, and return a special type that can be converted into a bool using `.is_true()`, `.is_false()`, or`.is_na()`. We could also use this sugar to write an efficient function to determine whether or not a numeric vector contains any missing values. To do this in R, we could use `any(is.na(x))`:

```
any_naR <- function(x) any(is.na(x))
```

However, this will do the same amount of work regardless of the location of the missing value. Here's the C++ implementation:

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


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

```
bool any_naC(NumericVector x) {
  return is_true(any(is_na(x)));
}
x0 <- runif(1e5)
x1 <- c(x0, NA)
x2 <- c(NA, x0)

microbenchmark(
  any_naR(x0), any_naC(x0),
  any_naR(x1), any_naC(x1),
  any_naR(x2), any_naC(x2)
)
#> Unit: microseconds
#>        expr   min     lq    mean median     uq    max neval
#>  any_naR(x0) 758.0 847.00  961.74 855.00 935.00  2,590   100
#>  any_naC(x0) 873.0 879.00  908.97 884.00 893.00  2,170   100
#>  any_naR(x1) 757.0 847.00  994.69 854.00 882.00  2,500   100
#>  any_naC(x1) 872.0 877.00  893.60 881.00 890.00  1,110   100
#>  any_naR(x2) 446.0 456.00 1232.36 463.00 475.00 58,700   100
#>  any_naC(x2)   3.6   4.79    6.72   6.09   7.78     16   100
```

Vector views

A number of helpful functions provide a “view” of a vector: head(), tail(), rep_each(), rep_len(), rev(),seq_along(), and seq_len(). In R these would all produce copies of the vector, but in Rcpp they simply point to the existing vector and override the subsetting operator ([) to implement special behaviour. This makes them very efficient: for instance, rep_len(x, 1e6) does not have to make a million copies of x.

Other useful functions

Finally, there’s a grab bag of sugar functions that mimic frequently used R functions:

- Math functions: abs(), acos(), asin(), atan(), beta(), ceil(), ceiling(), choose(), cos(), cosh(),digamma(), exp(), expm1(), factorial(), floor(), gamma(), lbeta(), lchoose(), lfactorial(), lgamma(),log(), log10

(), log1p(), pentagamma(), psigamma(), round(), signif(), sin(), sinh(), sqrt(), tan(), tanh(),tetragamma(), trigamma(), trunc().

- Scalar summaries: mean(), min(), max(), sum(), sd(), and (for vectors) var().
- Vector summaries: cumsum(), diff(), pmin(), and pmax().
- Finding values: match(), self_match(), which_max(), which_min().
- Dealing with duplicates: duplicated(), unique().
- d/q/p/r for all standard distributions.

Finally, noNA(x) asserts that the vector x does not contain any missing values, and allows optimisation of some mathematical operations.

The STL

The real strength of C++ shows itself when you need to implement more complex algorithms. The standard template library (STL) provides a set of extremely useful data structures and algorithms. This section will explain some of the most important algorithms and data structures and point you in the right direction to learn more. I can't teach you everything you need to know about the STL, but hopefully the examples will show you the power of the STL, and persuade you that it's useful to learn more.

If you need an algorithm or data structure that isn't implemented in STL, a good place to look is boost. Installing boost on your computer is beyond the scope of this chapter, but once you have it installed, you can use boost data structures and algorithms by including the appropriate header file with (e.g.) #include <boost/array.hpp>.

Using iterators

Iterators are used extensively in the STL: many functions either accept or return iterators. They are the next step up from basic loops, abstracting away the details of the underlying data structure. Iterators have three main operators:

1. Advance with ++.
2. Get the value they refer to, or **dereference**, with *.
3. Compare with ==.

For example we could re-write our sum function using iterators:

```
#include <Rcpp.h>

using namespace Rcpp;


// [[Rcpp::export]]

double sum3(NumericVector x) {

  double total = 0;
```

```
  NumericVector::iterator it;
  for(it = x.begin(); it != x.end(); ++it) {
    total += *it;
  }
  return total;
}
```

The main changes are in the for loop:

- We start at x.begin() and loop until we get to x.end(). A small optimization is to store the value of the end iterator so we don't need to look it up each time. This only saves about 2 ns per iteration, so it's only important when the calculations in the loop are very simple.
- Instead of indexing into x, we use the dereference operator to get its current value: *it.
- Notice the type of the iterator: NumericVector::iterator. Each vector type has its own iterator type:LogicalVector::iterator, CharacterVector::iterator, etc.

Iterators also allow us to use the C++ equivalents of the apply family of functions. For example, we could again rewritesum() to use the accumulate() function, which takes a starting and an ending iterator, and adds up all the values in the vector. The third argument to accumulate gives the initial value: it's particularly important because this also determines the data type that accumulate uses (so we use 0.0 and not 0 so that accumulate uses a double, not an int.). To useaccumulate() we need to include the <numeric> header.

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


// [[Rcpp::export]]
double sum4(NumericVector x) {
  return std::accumulate(x.begin(), x.end(), 0.0);
}
```

accumulate() (along with the other functions in <numeric>, like adjacent_difference(), inner_product(), andpartial_sum()) is not that important in Rcpp because Rcpp sugar provides equivalents.

Algorithms

The <algorithm> header provides a large number of algorithms that work with iterators. A good reference is available at http://www.cplusplus.com/reference/algorithm/. For example, we could write a basic Rcpp version of findInterval()that takes two arguments a vector of values and a vector of breaks,

and locates the bin that each x falls into. This shows off a few more advanced iterator features. Read the code below and see if you can figure out how it works.

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

// [[Rcpp::export]]
IntegerVector findInterval2(NumericVector x, NumericVector breaks) {
  IntegerVector out(x.size());

  NumericVector::iterator it, pos;
  IntegerVector::iterator out_it;

  for(it = x.begin(), out_it = out.begin(); it != x.end();
      ++it, ++out_it) {
    pos = std::upper_bound(breaks.begin(), breaks.end(), *it);
    *out_it = std::distance(breaks.begin(), pos);
  }

  return out;
}
```

The key points are:

- We step through two iterators (input and output) simultaneously.
- We can assign into an dereferenced iterator (`out_it`) to change the values in `out`.
- `upper_bound()` returns an iterator. If we wanted the value of the `upper_bound()` we could dereference it; to figure out its location, we use the `distance()` function.
- Small note: if we want this function to be as fast as `findInterval()` in R (which uses handwritten C code), we need to compute the calls to `.begin()` and `.end()` once and save the results. This is easy, but it distracts from this example so it has been omitted. Making this change yields a function that's slightly faster than R's`findInterval()` function, but is about 1/10 of the code.

It's generally better to use algorithms from the STL than hand rolled loops. In *Effective STL*, Scott Meyers gives three reasons: efficiency, correctness, and maintainability. Algorithms from the STL are written by C++ experts to be extremely efficient, and they have been around for a long time so they are

well tested. Using standard algorithms also makes the intent of your code more clear, helping to make it more readable and more maintainable.

Data structures

The STL provides a large set of data structures: array, bitset, list, forward_list, map, multimap, multiset,priority_queue, queue, dequeue, set, stack, unordered_map, unordered_set, unordered_multimap,unordered_multiset, and vector. The most important of these data structures are the vector, the unordered_set, and the unordered_map. We'll focus on these three in this section, but using the others is similar: they just have different performance trade-offs. For example, the deque (pronounced "deck") has a very similar interface to vectors but a different underlying implementation that has different performance trade-offs. You may want to try them for your problem. A good reference for STL data structures is http://www.cplusplus.com/reference/stl/ — I recommend you keep it open while working with the STL.

Rcpp knows how to convert from many STL data structures to their R equivalents, so you can return them from your functions without explicitly converting to R data structures.

Vectors

An STL vector is very similar to an R vector, except that it grows efficiently. This makes vectors appropriate to use when you don't know in advance how big the output will be. Vectors are templated, which means that you need to specify the type of object the vector will contain when you create it: vector<int>, vector<bool>, vector<double>,vector<String>. You can access individual elements of a vector using the standard [] notation, and you can add a new element to the end of the vector using .push_back(). If you have some idea in advance how big the vector will be, you can use .reserve() to allocate sufficient storage.

The following code implements run length encoding (rle()). It produces two vectors of output: a vector of values, and a vector lengths giving how many times each element is repeated. It works by looping through the input vector xcomparing each value to the previous: if it's the same, then it increments the last value in lengths; if it's different, it adds the value to the end of values, and sets the corresponding length to 1.

```
#include <Rcpp.h>

using namespace Rcpp;


// [[Rcpp::export]]

List rleC(NumericVector x) {

  std::vector<int> lengths;

  std::vector<double> values;


  // Initialise first value

  int i = 0;
```

```
  double prev = x[0];
  values.push_back(prev);
  lengths.push_back(1);

  NumericVector::iterator it;
  for(it = x.begin() + 1; it != x.end(); ++it) {
    if (prev == *it) {
      lengths[i]++;
    } else {
      values.push_back(*it);
      lengths.push_back(1);

      i++;
      prev = *it;
    }
  }

  return List::create(
    _["lengths"] = lengths,
    _["values"] = values
  );
}
```

(An alternative implementation would be to replace i with the iterator lengths.rbegin() which always points to the last element of the vector. You might want to try implementing that yourself.)

Other methods of a vector are described at http://www.cplusplus.com/reference/vector/vector/.

Sets

Sets maintain a unique set of values, and can efficiently tell if you've seen a value before. They are useful for problems that involve duplicates or unique values (like unique, duplicated, or in). C++ provides both ordered (std::set) and unordered sets (std::unordered_set), depending on whether or not order matters for you. Unordered sets tend to be much faster (because they use a hash table internally rather than a tree), so even if you need an ordered set, you should consider using an unordered set and then sorting the output. Like vectors, sets are templated, so you need to request the appropriate type of set for your purpose: unordered_set<int>, unordered_set<bool>, etc. More details are available

at http://www.cplusplus.com/reference/set/set/ andhttp://www.cplusplus.com/reference/unordered_set/unordered_set/.

The following function uses an unordered set to implement an equivalent to duplicated() for integer vectors. Note the use of seen.insert(x[i]).second. insert() returns a pair, the .first value is an iterator that points to element and the .second value is a boolean that's true if the value was a new addition to the set.

```
// [[Rcpp::plugins(cpp11)]]

#include <Rcpp.h>

#include <unordered_set>

using namespace Rcpp;


// [[Rcpp::export]]

LogicalVector duplicatedC(IntegerVector x) {

  std::unordered_set<int> seen;

  int n = x.size();

  LogicalVector out(n);


  for (int i = 0; i < n; ++i) {

    out[i] = !seen.insert(x[i]).second;

  }


  return out;

}
```

Note that unordered sets are only available in C++ 11, which means we need to use the cpp11 plugin,[[Rcpp::plugins(cpp11)]].

Map

A map is similar to a set, but instead of storing presence or absence, it can store additional data. It's useful for functions like table() or match() that need to look up a value. As with sets, there are ordered (std::map) and unordered (std::unordered_map) versions. Since maps have a value and a key, you need to specify both types when initialising a map: map<double, int>, unordered_map<int, double>, and so on. The following example shows how you could use a map to implement table() for numeric vectors:

```
#include <Rcpp.h>

using namespace Rcpp;
```

```
// [[Rcpp::export]]
std::map<double, int> tableC(NumericVector x) {
  std::map<double, int> counts;

  int n = x.size();
  for (int i = 0; i < n; i++) {
    counts[x[i]]++;
  }

  return counts;
}
```

Note that unordered maps are only available in C++ 11, so to use them, you'll again need [[Rcpp::plugins(cpp11)]].

Exercises

To practice using the STL algorithms and data structures, implement the following using R functions in C++, using the hints provided:

1. median.default() using partial_sort.
2. %in% using unordered_set and the find() or count() methods.
3. unique() using an unordered_set (challenge: do it in one line!).
4. min() using std::min(), or max() using std::max().
5. which.min() using min_element, or which.max() using max_element.
6. setdiff(), union(), and intersect() for integers using sorted ranges and set_union, set_intersection andset_difference.

Case studies

The following case studies illustrate some real life uses of C++ to replace slow R code.

Gibbs sampler

The following case study updates an example blogged about by Dirk Eddelbuettel, illustrating the conversion of a Gibbs sampler in R to C++. The R and C++ code shown below is very similar (it only took a few minutes to convert the R version to the C++ version), but runs about 20 times faster on my computer. Dirk's blog post also shows another way to make it even faster: using the faster random

number generator functions in GSL (easily accessible from R through the RcppGSL package) can make it another 2–3x faster.

The R code is as follows:

```
gibbs_r <- function(N, thin) {
  mat <- matrix(nrow = N, ncol = 2)
  x <- y <- 0

  for (i in 1:N) {
    for (j in 1:thin) {
      x <- rgamma(1, 3, y * y + 4)
      y <- rnorm(1, 1 / (x + 1), 1 / sqrt(2 * (x + 1)))
    }
    mat[i, ] <- c(x, y)
  }
  mat
}
```

This is straightforward to convert to C++. We:

- add type declarations to all variables
- use (instead of [to index into the matrix
- subscript the results of rgamma and rnorm to convert from a vector into a scalar

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

// [[Rcpp::export]]
NumericMatrix gibbs_cpp(int N, int thin) {
  NumericMatrix mat(N, 2);
  double x = 0, y = 0;

  for(int i = 0; i < N; i++) {
    for(int j = 0; j < thin; j++) {
      x = rgamma(1, 3, 1 / (y * y + 4))[0];
```

```
      y = rnorm(1, 1 / (x + 1), 1 / sqrt(2 * (x + 1)))[0];
    }
    mat(i, 0) = x;
    mat(i, 1) = y;
  }

  return(mat);
}
```

Benchmarking the two implementations yields:

```
microbenchmark(
  gibbs_r(100, 10),
  gibbs_cpp(100, 10)
)
#> Unit: microseconds
#>              expr    min     lq  mean median     uq    max neval
#>    gibbs_r(100, 10) 14,700 17,200 17364 17,300 17,400 21,500   100
#>  gibbs_cpp(100, 10)   636    671   689    687    702    844   100
```

R vectorisation vs. C++ vectorisation

This example is adapted from "Rcpp is smoking fast for agent-based models in data frames". The challenge is to predict a model response from three inputs. The basic R version of the predictor looks like:

```
vacc1a <- function(age, female, ily) {
  p <- 0.25 + 0.3 * 1 / (1 - exp(0.04 * age)) + 0.1 * ily
  p <- p * if (female) 1.25 else 0.75
  p <- max(0, p)
  p <- min(1, p)
  p
}
```

We want to be able to apply this function to many inputs, so we might write a vector-input version using a for loop.

```
vacc1 <- function(age, female, ily) {
  n <- length(age)
  out <- numeric(n)
  for (i in seq_len(n)) {
    out[i] <- vacc1a(age[i], female[i], ily[i])
  }
  out
}
```

If you're familiar with R, you'll have a gut feeling that this will be slow, and indeed it is. There are two ways we could attack this problem. If you have a good R vocabulary, you might immediately see how to vectorise the function (using `ifelse()`, `pmin()`, and `pmax()`). Alternatively, we could rewrite `vacc1a()` and `vacc1()` in C++, using our knowledge that loops and function calls have much lower overhead in C++.

Either approach is fairly straightforward. In R:

```
vacc2 <- function(age, female, ily) {
  p <- 0.25 + 0.3 * 1 / (1 - exp(0.04 * age)) + 0.1 * ily
  p <- p * ifelse(female, 1.25, 0.75)
  p <- pmax(0, p)
  p <- pmin(1, p)
  p
}
```

(If you've worked R a lot you might recognise some potential bottlenecks in this code: `ifelse`, `pmin`, and `pmax` are known to be slow, and could be replaced with `p + 0.75 + 0.5 * female`, `p[p < 0] <- 0`, `p[p > 1] <- 1`. You might want to try timing those variations yourself.)

Or in C++:

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

double vacc3a(double age, bool female, bool ily){
  double p = 0.25 + 0.3 * 1 / (1 - exp(0.04 * age)) + 0.1 * ily;
  p = p * (female ? 1.25 : 0.75);
  p = std::max(p, 0.0);
```

```
  p = std::min(p, 1.0);
  return p;
}

// [[Rcpp::export]]
NumericVector vacc3(NumericVector age, LogicalVector female,
                    LogicalVector ily) {
  int n = age.size();
  NumericVector out(n);

  for(int i = 0; i < n; ++i) {
    out[i] = vacc3a(age[i], female[i], ily[i]);
  }

  return out;
}
```

We next generate some sample data, and check that all three versions return the same values:

```
n <- 1000
age <- rnorm(n, mean = 50, sd = 10)
female <- sample(c(T, F), n, rep = TRUE)
ily <- sample(c(T, F), n, prob = c(0.8, 0.2), rep = TRUE)

stopifnot(
  all.equal(vacc1(age, female, ily), vacc2(age, female, ily)),
  all.equal(vacc1(age, female, ily), vacc3(age, female, ily))
)
```

The original blog post forgot to do this, and introduced a bug in the C++ version: it used 0.004 instead of 0.04. Finally, we can benchmark our three approaches:

```
microbenchmark(
  vacc1 = vacc1(age, female, ily),
```

```
  vacc2 = vacc2(age, female, ily),
  vacc3 = vacc3(age, female, ily)
)
#> Unit: microseconds
#>   expr     min      lq   mean  median    uq    max neval
#>  vacc1 7,940.0 8,170.0 9266.4 8,300.0 8,710 67,300   100
#>  vacc2   389.0   401.0  458.1   437.0   474  2,080   100
#>  vacc3    61.1    62.8   69.9    68.5    74    103   100
```

Not surprisingly, our original approach with loops is very slow. Vectorising in R gives a huge speedup, and we can eke out even more performance (~10x) with the C++ loop. I was a little surprised that the C++ was so much faster, but it is because the R version has to create 11 vectors to store intermediate results, where the C++ code only needs to create 1.

Using Rcpp in a package

The same C++ code that is used with sourceCpp() can also be bundled into a package. There are several benefits of moving code from a stand-alone C++ source file to a package:

1. Your code can be made available to users without C++ development tools.
2. Multiple source files and their dependencies are handled automatically by the R package build system.
3. Packages provide additional infrastructure for testing, documentation, and consistency.

To add Rcpp to an existing package, you put your C++ files in the src/ directory and modify/create the following configuration files:

- In DESCRIPTION add

```
LinkingTo: Rcpp
```

```
Imports: Rcpp
```

- Make sure your NAMESPACE includes:

```
useDynLib(mypackage)
```

```
importFrom(Rcpp, sourceCpp)
```

We need to import something (anything) from Rcpp so that internal Rcpp code is properly loaded. This is a bug in R and hopefully will be fixed in the future.

To generate a new Rcpp package that includes a simple “hello world” function you can use Rcpp.package.skeleton():

```
Rcpp.package.skeleton("NewPackage", attributes = TRUE)
```

To generate a package based on C++ files that you've been using with sourceCpp(), use the cpp_files parameter:

```
Rcpp.package.skeleton("NewPackage", example_code = FALSE,
                      cpp_files = c("convolve.cpp"))
```

Before building the packge, you'll need to run Rcpp::compileAttributes(). This function scans the C++ files forRcpp::export attributes and generates the code required to make the functions available in R. Re-runcompileAttributes() whenever functions are added, removed, or have their signatures changed. This is done automatically by the devtools package and by Rstudio.

For more details see the Rcpp package vignette, vignette("Rcpp-package").

Learning more

This chapter has only touched on a small part of Rcpp, giving you the basic tools to rewrite poorly performing R code in C++. The Rcpp book is the best reference to learn more about Rcpp. As noted, Rcpp has many other capabilities that make it easy to interface R to existing C++ code, including:

- Additional features of attributes including specifying default arguments, linking in external C++ dependencies, and exporting C++ interfaces from packages. These features and more are covered in the Rcpp attributes vignette, vignette("Rcpp-attributes").
- Automatically creating wrappers between C++ data structures and R data structures, including mapping C++ classes to reference classes. A good introduction to this topic is Rcpp modules vignette, vignette("Rcpp-modules")
- The Rcpp quick reference guide, vignette("Rcpp-quickref"), contains a useful summary of Rcpp classes and common programming idioms.

I strongly recommend keeping an eye on the Rcpp homepage and Dirk's Rcpp page as well as signing up for the Rcpp mailing list. Rcpp is still under active development, and is getting better with every release.

Other resources I've found helpful in learning C++ are:

- *Effective C++* and *Effective STL* by Scott Meyers.
- *C++ Annotations*, aimed at "knowledgeable users of C (or any other language using a C-like grammar, like Perl or Java) who would like to know more about, or make the transition to, C++".
- *Algorithm Libraries*, which provides a more technical, but still concise, description of important STL concepts. (Follow the links under notes).

Writing performant code may also require you to rethink your basic approach: a solid understanding of basic data structures and algorithms is very helpful here. That's beyond the scope of this book, but I'd suggest the *Algorithm Design Manual*, MIT's *Introduction to Algorithms*, *Algorithms* by Robert Sedgewick and Kevin Wayne which has a freeonline textbook and a matching coursera course.

Acknowledgments

I’d like to thank the Rcpp-mailing list for many helpful conversations, particularly Romain Francois and Dirk Eddelbuettel who have not only provided detailed answers to many of my questions, but have been incredibly responsive at improving Rcpp. This chapter would not have been possible without JJ Allaire; he encouraged me to learn C++ and then answered many of my dumb questions along the way.