

# Statistics 360: Advanced R for Data Science

## Lecture 10

Becky Lin



## References:

- ▶ Chapter 25 of Advanced R by Wickham.
- ▶ The Rcpp website: <http://www.rcpp.org/>
- ▶ Rcpp gallery: <https://gallery.rcpp.org/>
- ▶ Rcpp quick reference  
<https://dirk.eddelbuettel.com/code/rcpp/Rcpp-quickref.pdf>

# Calling C++ from R with Rcpp

- ▶ R is written in C, and so in principle it is possible to write C/C++ code that calls R and R code that calls C/C++.
  - ▶ Using R “internals” directly is complicated.
  - ▶ A base R function for passing R objects to C/C++ is `.Call`.
  - ▶ See the chapter “System and foreign language interfaces” in the “Writing R Extensions” manual.
- ▶ Rcpp provides a more user-friendly interface with C++.
  - ▶ Allows you to write C++ functions that can be called from R
  - ▶ Use it to speed up code that runs too slowly in R.
  - ▶ Of the 17345 packages on CRAN, 2385 (or more) use Rcpp
- ▶ This is a brief intro; much more info at <http://www.rcpp.org/> and elsewhere on the internet.
- ▶ Warning: This lecture will be too simplified for those with C++ experience, and information overload for those without.

# Use cases

- ▶ Unavoidable loops (can't avoid by vectorizing).
- ▶ Many (e.g., millions) of function calls, such as in a recursive algorithm.
  - ▶ Less overhead in C++ compared to R.
- ▶ Require advanced data structures and algorithms that are available in a C++ library, such as the Standard Template Library( STL), but not in R.
- ▶ And many more . . .

# Prerequisites

- ▶ Install and load the Rcpp package

```
# install.packages("Rcpp")  
library(Rcpp)
```

- ▶ Install a working C++ compiler. To get it:
  - ▶ On Windows, install Rtools.
  - ▶ On Mac, install Xcode.
  - ▶ On Linux, `sudo apt-get install r-base-dev`

## Getting started (section 25.2)

- ▶ First example:

```
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)  
## .Call(<pointer: 0x10bc83850>, x, y, z)
```

```
add(1, 2, 3)
```

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

- ▶ Rcpp (i) compiles the C++ code (note lag) and (ii) constructs the R function `add()` that will call this compiled code.
- ▶ The above defines a function “inline”; it is also possible to “source” C++ code (more later).

# The plan

- ▶ Start simple and work up, writing functions with:
  - ▶ no inputs and a scalar output
  - ▶ scalar input and scalar output
  - ▶ vector input and scalar output
  - ▶ vector input and vector output
  - ▶ matrix input and vector output
- ▶ Keep an eye out for differences, such as the need to declare the type of objects.



## No inputs, scalar output

► R:

```
one <- function() 1L # recall that 1L is integer 1
```

► Rcpp (notice single quotes):

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

```
one()
```

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

## C++ differences

- ▶ Don't use assignment to create functions.
- ▶ Declare the type of output the function returns; e.g., a scalar integer (`int`).
- ▶ C++ has true scalars, with types `double`, `int`, `String`, and `bool`.
- ▶ A return statement is **required**
- ▶ Every statement is terminated by a `;`.

## Scalar input and output

- A scalar version of the `sign()` function which returns 1, 0 or  $-1$  for positive, zero or negative input:

```
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;  
  }  
}')  
signC(-100)
```

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

# Notes

- ▶ In addition to declaring the type of the output, we must declare the type of the input.
- ▶ Logicals and `if-else` are the same.

## Vector input and scalar output

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

```
set.seed(1)
x <- rnorm(1e5)
bench::mark(sumR(x), sumC(x), sum(x)) # recall benchmarking pkg be
```

```
## # A tibble: 3 x 6
```

##	expression	min	median	`itr/sec`	mem_alloc	`gc/sec`
##	<bch:expr>	<bch:tm>	<bch:tm>	<dbl>	<bch:byt>	<dbl>
## 1	sumR(x)	4.41ms	4.48ms	223.	3.99MB	0
## 2	sumC(x)	94.17us	95.08us	10440.	2.49KB	0
## 3	sum(x)	11.38ms	11.48ms	87.1	0B	0

# Notes

- ▶ The input in this example is of type `NumericVector`, which is a C++ class defined by `Rcpp`.
  - ▶ `Rcpp` takes care of converting the R function input to `NumericVector` and the C++ function output to an R object
  - ▶ Other vector types are `IntegerVector`, `CharacterVector`, and `LogicalVector`.
- ▶ The `.size()` method of a vector returns the length as an integer.
- ▶ Notice the syntax of `for()`: `for(initial condn; check condn; increment)`.
  - ▶ In this case we initialise by creating variable `i` with value 0.
  - ▶ Before each iteration we check that `i < n`, and terminate if not.
  - ▶ After each iteration, increment `i` by one, using `++`.
- ▶ C++ uses zero-based indexing, `0, ... n-1` (!!!)
- ▶ `+=` increments “in-place”

## Vector input, vector output

- Note: In the following, `NumericVector out (n)` constructs a `NumericVector` object of length `n`.

```
pdistR <- function(x, ys) { sqrt((x - ys) ^ 2) }  
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)); // pow() vs ^{}  
  }  
  return out;  
}')  
pdistC(10,6:15)
```

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

```
try(pdistC(1:3, 6:15) )
```

```
## Error in pdistC(1:3, 6:15) : Expecting a single value: [extent=3].
```



## Copying with clone

- ▶ In R we use assignment to copy vectors.
- ▶ In C++ with Rcpp copy an existing vector with the `clone()` function; e.g., `NumericVector zs = clone(ys)`.

## Using sourceCpp

- ▶ For functions of more than a few lines it is more convenient to define them in a source file and use `sourceCpp()` to link them to R.
- ▶ Source files must end in `.cpp` and start with the header

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

- ▶ The File -> New File -> C++ file feature of RStudio generates a skeleton to get you started; see `lec10_1.cpp`.

```
sourceCpp("lec10_1.cpp") # See source file
```

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

```
timesTwo(3:6)
```

```
## [1] 6 8 10 12
```

## Other classes (Section 25.3)

- ▶ Rcpp provides wrappers to other base data type.
- ▶ The text focusses on lists/data frames, functions and attributes.

## List input, including S3 classes

- ▶ Generally more useful for output than input, because C++ needs to know classes of list components in advance.
- ▶ For use as input, you can convert components to C++ equivalents with `as()`.
  - ▶ See the source file `lec10_2.cpp` for the following example.

```
sourceCpp("lec10_2.cpp")
mod <- lm(mpg ~ wt, data = mtcars)
mpe(mod) # function defined in lec10_2.cpp
```

```
## [1] -0.01541615
```

## Aside: Primer on C++ templates

- ▶ Few details here, just a few words so we recognize function and class templates when we see them.
- ▶ C++ template functions allow programmers to write code that takes a generic argument.
  - ▶ A template function `func<T>()` takes template parameter `T` that can be the type of the function inputs and/or output. For a given `T`, the compiler generates a function specific to that type.
  - ▶ Examples: `as<NumericVector>(SEXP R)` is a function that takes an R expression (`SEXP`) as input and returns a `NumericVector`; `as<CharacterVector>(SEXP R)` returns a `CharacterVector`; etc.
- ▶ Data structures can also be templated; e.g., the C++ Standard Template Library (STL) defines a `vector` class that is like a generic container, with methods for inserting, re-sizing, etc.
  - ▶ You can create a vector of `ints` with `vector<int>`, etc.
- ▶ Defining your own template functions and classes is beyond the scope of this class.

# Functions

- ▶ So far we have been using Rcpp to call C++ functions from R, but we can also call R functions from C++.
- ▶ Use type Function to input R functions and type RObject to hold general input/output.

```
sourceCpp("lec10_3.cpp") # see source file
set.seed(123)
x <- rnorm(100)
callFunction(x,fivenum)
```

```
## [1] -2.30916888 -0.49667731  0.06175631  0.69499808  2.18733299
```

```
callWithOne(function(x) x+1)
```

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

```
fit <- lm_in_C(formula(mpg~disp),mtcars,lm)
summary(fit)$coef
```

```
##              Estimate Std. Error  t value    Pr(>|t|)
## (Intercept) 29.59985476 1.229719515 24.070411 3.576586e-21
## disp        -0.04121512 0.004711833 -8.747152 9.380327e-10
```

## A warning from the Rcpp developers

- ▶ Occasional R function calls from C++ are OK, but don't call repeatedly.
- ▶ From <https://gallery.rcpp.org/articles/r-function-from-c++/> :  
*Calling a function is simple and tempting. It is also slow as there are overheads involved. And calling it repeatedly from inside your C++ code, possibly buried within several loops, is outright silly. This has to be slower than equivalent C++ code, and even slower than just the R code (because of the marshalling of data). Do it when it makes sense, and not simply because it is available.*
- ▶ Consider instead accessing R functionality from the GNU Scientific Library  
<https://www.gnu.org/software/gsl/doc/html/index.html> via the RcppGSL package, or one of the 50 or so other Rcpp\* C/C++ library interfaces.

# Attributes

- In R we get and set attributes with `attr()`; e.g.,

```
out <- c(1,2,3)
names(out) <- c("a","b","c")
attr(out,"my-attr") <- "my-value"
attr(out,"class") <- "my-class" # or class(out) <- "my-class"
out
```

```
## a b c
## 1 2 3
## attr(,"my-attr")
## [1] "my-value"
## attr(,"class")
## [1] "my-class"
```



# Attributes with Rcpp

- From Rcpp, get and set R object attributes with the `.names()` and `.attr()` methods for vector types.

```
sourceCpp("lec10_4.cpp") # see source file
attribs()
```

```
## a b c
## 1 2 3
## attr("my-attr")
## [1] "my-value"
## attr("class")
## [1] "my-class"
```

## C++ Standard Template Library (Section 25.5)

- ▶ References: Text, section 25.5, and [https://www.cppreference.com/Cpp\\_STL\\_ReferenceManual.pdf](https://www.cppreference.com/Cpp_STL_ReferenceManual.pdf)
- ▶ Rcpp gives us access to the data structures and algorithms provided by C++ libraries like the Standard Template Library (STL).
- ▶ We'll cover some STL basics.
  - ▶ Data structures
  - ▶ Iterators
  - ▶ Algorithms
- ▶ The theme of the STL is abstraction: Implementations of algorithms encapsulate the logic and operate on generic data structures that we think of as “containers”, capable of holding any kind of data.
- ▶ Note: Many of the data structures and algorithms that originated in the STL have made their way into the C++ Standard Library; this is true of all of those used in this lecture.

# Lists in R are generic containers

- ▶ In R, lists can be thought of as generic containers that generalize atomic vectors to hold arbitrary data structures.

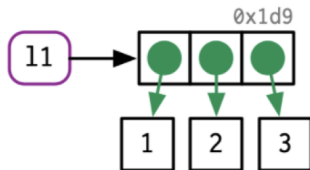


Figure 1: Lists as generic containers

- ▶ In R, list elements can be accessed by index, in the same way as vectors (in fact, we refer to lists as vectors); this is called “random access”.

# Container performance tradoffs

- ▶ The STL offers different containers with different performance characteristics. For example:
  - ▶ vectors allow fast random access, but are slow for insertion/deletion
  - ▶ lists (not R lists) allow fast insertion/deletion, but access to elements requires traversal of the container, starting at one end.
- ▶ Unlike R lists, STL vector and list elements must be of the same type.
  - ▶ E.G., create a vector of integers of length 9 with `vector<int> myvec (9)`
  - ▶ Rcpp data structures like `NumericVector` are essentially typed STL vectors (`NumericVector` is `vector<double>`).
- ▶ Algorithms can be written to work on multiple container types by abstracting how we access elements. The abstraction is called an iterator.

# Using iterators

- ▶ We will use the following three features of iterators:
  1. Advance with ++ or --.
  2. Get the value they refer to, or dereference, with \*.
  3. Compare with == and !=.
- ▶ The following code snippets compare a sum function that uses indexing to one that uses an iterator to loop over the vector.

```
#include <Rcpp.h>
using namespace Rcpp;
// [[Rcpp::export]]
double sumC(NumericVector x) {
    int n = x.size();
    double total = 0;
    for(int i = 0; i < n; ++i) {
        total += x[i];
    }
    return total;
}
// [[Rcpp::export]]
double sum3(NumericVector x) {
    double total = 0;
    NumericVector::iterator it;
    for(it = x.begin(); it != x.end(); ++it) {
        total += *it;
    }
    return total;
}
```

- ▶ At any given time, the iterator will “refer” (point) to an element of the container.
- ▶ We initialize to refer to the first element of the container `x` with `x.begin()` and will stop iterating when the iterator refers to the last element `x.end()`.
- ▶ `++it` advances the iterator to refer to the next element of the container.
- ▶ `*it` gets the container element that it currently refers to.

# Algorithms

- ▶ Algorithms in the STL often input or output iterators.
- ▶ The script `lec10_5.cpp` gives an example from the text.
  - ▶ Implements R's `findInterval()` function that takes a numeric vector `x` and vector of breakpoints `breaks` as input and returns the bin defined by `breaks` that each element of `x` is in.

```
sourceCpp("lec10_5.cpp")  
findInterval2(x=c(-1.5,-.5,.5,1.5),breaks=c(-1,0,1))
```

```
## [1] 0 1 2 3
```

- ▶ Iterates over input `x` with iterator `it` and over output vector `out` with iterator `out_it`.
  - ▶ Call `upper_bound()` algorithm from the STL (also in C++ Std Lib) to search breaks for the first element greater than `*it`. `upper_bound()` returns an iterator `pos` over the vector breaks.
  - ▶ Calculate bin (interval) number as distance from start of breaks to `pos`.



## Case Studies (Section 25.6)

- ▶ See the Gibbs sampler and vectorization examples in Section 25.6 of the text
- ▶ We'll do a different case study.

## Case Study: Metropolis algorithm

- ▶ The Metropolis-Hastings (MH) algorithm is an important tool for drawing (dependent) samples from a distribution known only up to a constant.
  - ▶ A Markov chain Monte Carlo (MCMC) algorithm that reinvigorated Bayesian statistics in the early 2000s.
  - ▶ We will implement a simplified version of the MH algorithm known as the Metroplois algorithm.
- ▶ Reference for this demo is Matthew Stephens' "Five Minute Stats" site:
  - ▶ Introduction to the Metropolis-Hastings algorithm  
[https://stephens999.github.io/fiveMinuteStats/MH\\_intro.html](https://stephens999.github.io/fiveMinuteStats/MH_intro.html)
  - ▶ Example implementations in R  
<https://stephens999.github.io/fiveMinuteStats/MH-examples1.html>

## Example Metropolis algorithm

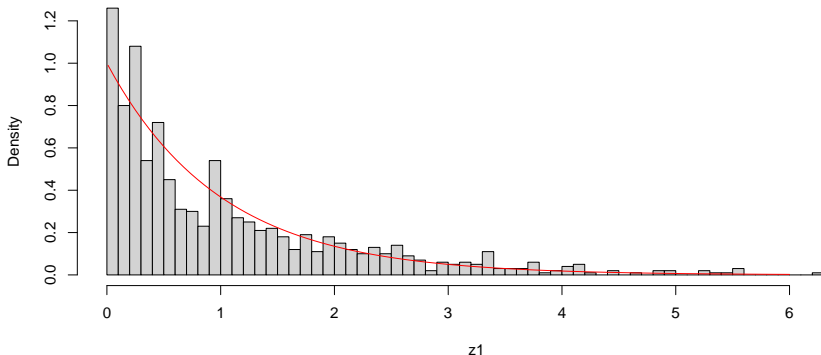
- ▶ Problem: sample from an exponential distribution with mean 1.
  - ▶ Simple illustration – we know how to do this without MCMC.
- ▶ R code to implement the algorithm is as follows

```
target = function(x){  
  if(x<0){ return(0) } else { return( exp(-x))}  
}  
easyMCMC_R = function(niter, startval, proposalsd){  
  x = rep(0,niter)  
  x[1] = startval  
  for(i in 2:niter){  
    currentx = x[i-1]  
    proposedx = rnorm(1,mean=currentx,sd=proposalsd)  
    A = target(proposedx)/target(currentx)  
    if(runif(1)<A){  
      x[i] = proposedx #accept move with probability min(1,A)  
    } else {  
      x[i] = currentx #otherwise "reject" move, and stay where we are  
    }  
  }  
  return(x)  
}
```

# Test out easyMCMC

```
set.seed(123)
N <- 1000; startval <- 3; proposalsd <- 1
z1=easyMCMC_R(N,startval,proposalsd)
hist(z1,nclass=50,freq=FALSE,xlim=c(0,6.5))
xx <- seq(from=0.01,to=6,length=100)
lines(xx,exp(-xx),col="red")
```

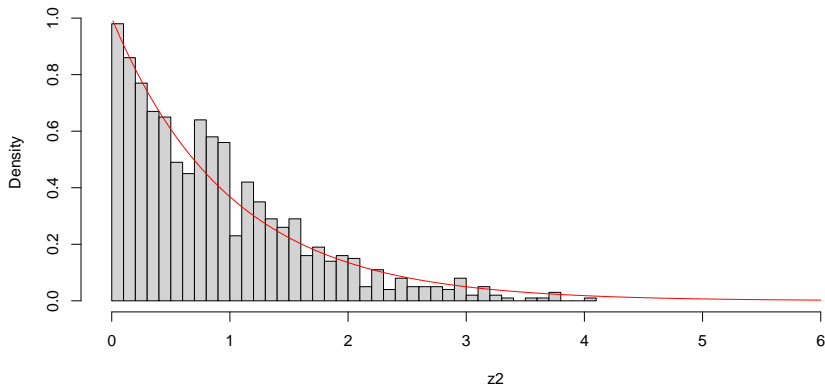
Histogram of z1



## Rcpp version

```
sourceCpp("lec10_MCMC.cpp")  
z2 <- easyMCMC_C(N,startval,proposalsd)  
hist(z2,nclass=50,freq=FALSE,xlim=c(0,6.5))  
lines(xx,exp(-xx),col="red")
```

Histogram of z2



# Benchmark the two implementations

- ▶ The C++ version runs about 20 times faster.
- ▶ For more complex problems, MCMC algorithms need to run for millions of iterations, so 20-times speedup is good (think one day vs three weeks run-time).

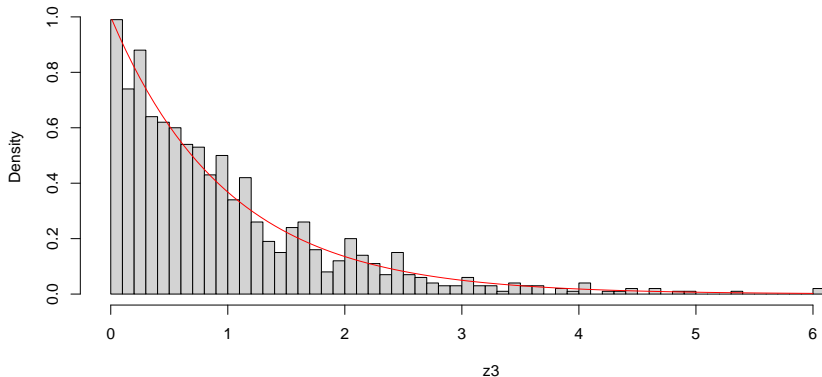
```
bench::mark(easyMCMC_R(N,startval,proposalsd),  
            easyMCMC_C(N,startval,proposalsd),  
            check=FALSE) # don't check equality of function calls
```

```
## # A tibble: 2 x 6  
##   expression          min    median `itr/sec` mem_alloc  
##   <bch:expr>      <bch:tm> <bch:tm>      <dbl> <bch:byt>  
## 1 easyMCMC_R(N, startval, proposalsd)  3.18ms   3.49ms    275.    4.87MB  
## 2 easyMCMC_C(N, startval, proposalsd) 211.96us  217.5us  4464.   14.48KB  
## # ... with 1 more variable: `gc/sec` <dbl>
```

# Compare to R's `rexp()` random exponential generator

```
z3 <- rexp(1000)
hist(z3, nclass=50, freq=FALSE, xlim=c(0, 6.5))
lines(xx, exp(-xx), col="red")
```

Histogram of z3



## Rcpp in R packages (Section 25.7)

- ▶ Your .cpp files can also be included in a package.
- ▶ Call `devtools::use_rcpp()` from your package's RStudio project to get started
  - ▶ Compiled code goes in a `src` directory.
  - ▶ Rcpp is added to the “Imports” and “LinkingTo” fields of the DESCRIPTION file.
  - ▶ And more ...

```
> devtools::use_rcpp()
Creating 'src/'
Adding '*.o', '*.so', '*.dll' to 'src/.gitignore'
Copy and paste the following lines into 'R/mars-package.R':
## usethis namespace: start
#' @useDynLib mars, .registration = TRUE
## usethis namespace: end
NULL
[Copied to clipboard]
Adding 'Rcpp' to LinkingTo field in DESCRIPTION
Adding 'Rcpp' to Imports field in DESCRIPTION
Copy and paste the following lines into 'R/mars-package.R':
## usethis namespace: start
#' @importFrom Rcpp sourceCpp
## usethis namespace: end
NULL
[Copied to clipboard]
Writing 'src/code.cpp'
Modify 'src/code.cpp'
```



## Topics not covered

- ▶ Missing values (Section 25.4)