uc3m | Universidad Carlos III de Madrid

OPENCOURSEWARE
ADVANCED PROGRAMMING
STATISTICS FOR DATA SCIENCE

Ricardo Aler

RCPP

Seamless R and C++ integration

Instalation and test

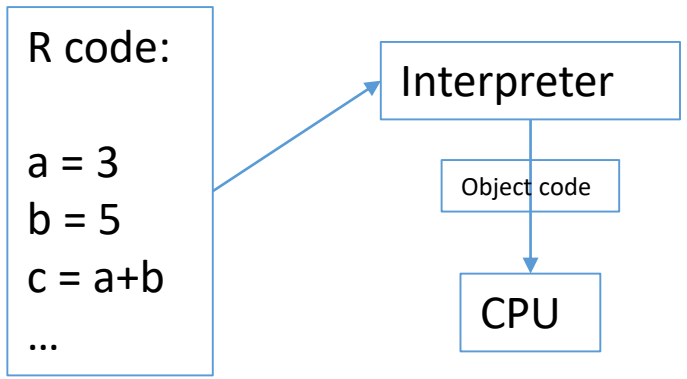
```
install.packages("Rcpp")
library(Rcpp)
```

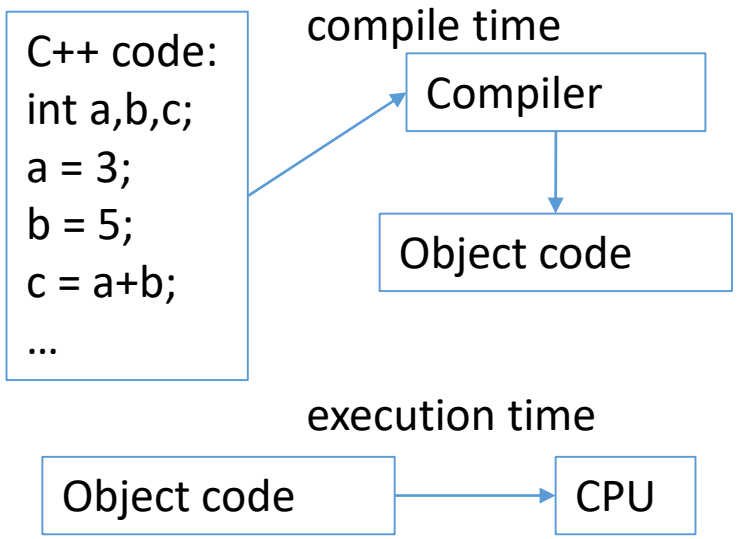
Why Rcpp?

- Rcpp extends the R language using the C++ language
- Motivation:
  - Speed: R is an interpreted language (slow) vs. C++ is a compiled language (fast)

Interpreted (R) vs. Compiled (C++)

- Interpreted:
  - The R interpreter executes source code line by line
  - It is slow because, among other reasons, the interpreter has to carry out lots of checkings at run-time (i.e. when the program is running)
  - But they are usually more flexible.
- Compiled:
  - Compiles source code into object code (machine code), that can be run directly by the CPU. Most checkings can be done at compile-time (i.e. when the program is compiled), just once, and not everytime the program is run (at run-time or execution-time, like in R). Therefore increasing speed.
  - But they are usually less flexible.





Dynamic typing (R) vs. Static typing (C++)

- In R variables can contain objects of any type:

```
# Variable x contains a string of characters
x <- "In a hole in the ground there lived a hobbit"
# Now variable x contains an integer
x <- 3
```

  - This is time consuming. In order to run the two previous lines, the R interpreter has to:
    1. Check if there is space in memory for the string
    2. Allocate space
    3. Point x to that space
    4. Check if there is space in memory for the integer
    5. Allocate space in memory
    6. Point x to that space
- In C++, variables must have a type and can only contain objects belonging to that type
  - For instance, if we declare x as being integer (like this: *int x;*), then C++ already knows that integer variables take 2 units of memory to be stored (2 bytes), and reserves that amount of space. Thus, it does not have to check everytime whether there is enough space because it is already reserved. Hence, <u>saving time.</u>

Dynamic typing (R) vs. Static typing (C++)

- In C++, variables must have a type and can only contain objects belonging to that type
  - It is also necessary to specify in advance the object type returned by a function
- *cppFunction* compiles the C++ code contained in the string
- Notice other differences with R code:
  - Comments use // (instead of #)
  - Sentences end in ;

```
library(Rcpp)

cppFunction(
'int my_function() {
  // Variable declaration
  int x;

  // Use of the variable
  x = 3;

  return(x);
}'
)

my_function()
```

```
[1] 3
```

When R is slow

- R is particularly slow:
  - When using loops (for, while, …)
  - When having to call functions lots of times

- C++ can speed the code a lot

Comparing average execution time for a function that adds all elements in a vector: *sum(c(1,2,3))==6*

C++

```
double sumC(NumericVector x) {

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

R

```
sumR <- function(x) {
  n <- length(x)
  total <- 0
  for (i in 1:n) {
    total <- total + x[i]
  }
  total
}
```

sumR(c(1,2,3))
[1] 6

Median execution time for: x <- **runif**(1e6)

sum(x): 712 microseconds
sumC(x): 942 microseconds
sumR(x): 30389 microseconds. R's loop is 42 times slower than C's!!!

Further reading on C++

- Brokken FB (2012) C++ annotations. Electronic book, University of Groningen,
  - http://www.icce.rug.nl/documents/cplusplus

Some examples using Rcpp

- (more information in http://adv-r.had.co.nz/Rcpp.html)

- In order to learn Rcpp, we are going to convert some functions, defined in R, into C++
  - We'll see the code is very similar
- The way I'm going to teach C++ is by means of examples
- Index:
  1. No inputs, scalar output
  2. Scalar input, scalar output
  3. Vector input, scalar output
  4. Vector input, vector output

Function definition in R and C++

- In R, function definitions is like:

In R

```r
my_function <- function(y) {

# No argument type declaration
# No Variable declaration
# Use of the variable
  x = 1+y
# The last evaluation is returned
  x
}
```

output type declaration

argument type declaration

In C++

```cpp
int my_function (int y){
  // Variable declaration
  int x;
  // Use of the variable
  x = 1+y;
  // Use return
  return(x);
}
```

1. No inputs, scalar output

- Let's define a function that returns "1"
- In R it wold be:

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

- In C++:

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

- Notice that:
  - We must specify the type of the output of the function (int)
  - Sentences end in ;
  - We must use return explicitely in order to return a value

No inputs, scalar output

- In order to use it in R, the complete code would be:

```
library(Rcpp)

# cppFunction compiles the C++ function
cppFunction('
int one() {
  return(1);
}
')

# And now, we can use it in R

x <- one()
print(x)
```

```
[1] 1
```

Scalar input, scalar output

- Let's define a function that returns the sign of an integer number

In R

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

In C++

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

- Arguments of functions must have a type (int x)
- The syntax of if-then-else is similar to R, but not exactly the same

EXERCISES 1.

- Write your own my_mean function that computes the mean of a vector
- It takes an integer vector of real numbers as input, and returns a real value

```
library(Rcpp)

#####################################
# my_mean()
#####################################

cppFunction('
double my_mean(NumericVector x) {
< WRITE YOUR CODE HERE>
return y;
}
')
```

test it with in R with:

```
my_mean(c(1,2,3))
[1] 2
```

Vector input, scalar output

- Let's define in C++ a function that adds all the values in a vector of integers. E.g.: sum(c(2,4,6,8)) : 20
- In R, this function already exists: *sum()*
- But we can also rewrite the function in R, using R loops. Let's do it to compare C++ loops and R loops:

In R

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

Vector input, scalar output

In R

```
sumR <- function(x) {
  n = length(x)
  total <- 0
  for (i in 1:n) {
    total <- total + x[i]
  }
  total
}
```

Note: In R, in order to handle the null x case, it's better to write for(i in seq_along(x) { …

In C++

```
double sumC(NumericVector x) {

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

- NumericVector is the type of the input
- double is the type of the ouput
  - Double = real number
- for is a loop, similar to R's
  - for(init; check; increment)
- .size() is a C++ method.
  - In R, we would write: length(x)
- In C++, INDICES START AT 0!!
  - Unlike R, where they start at 1

Vector input, scalar output

In R

```
sumR <- function(x) {
  n = length(x)
  total <- 0
  for (i in 1:n) {
    total <- total + x[i]
  }
  total
}
```

In C++

```
double sumC(NumericVector x) {

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

Median execution time for: x <- **runif**(1e6)

sum(x): 712 microseconds
sumC(x): 942 microseconds
sumR(x): 30389 microseconds. R's loop is 42 times slower than C's!!!

Vector input, vector output

- Let's define a function that checks whether values in a vector are smaller or equal than some constant. If they are, the element is changed to zero.
- Example:

```
input <- c(1,2,3,4,3,2,1)
my_check(input, 2)

[1] 0 0 3 4 3 0 0
```

- We can do this in R in two ways:

Using R functions (ifelse)

```
my_check_R1 <- function(x, c){
  ifelse(x<=c, 0, x)
}

[1] 0 0 3 4 3 0 0
```

Using R loops

```
my_check_R2 <- function(x, c){
  out = x
  n = length(x)
  for(i in 1:n){
    if(x[i]<=c) {out[i] = 0}
  }
}

[1] 0 0 3 4 3 0 0
```

In R

```r
my_check_R2 <- function(x, c){
  n = length(x)
  out = x
  for(i in 1:n){
    if(x[i]<=c) {
      out[i] = 0
    }
  }
  # Vector out is returned
  out
}
```

Median execution time (microseconds)
for input = sample(10, 1e6, replace=TRUE)

my_check_R1(input, 2): 41 ms
my_check_R2(input, 2): 42 ms
my_check_C(input, 2): 4 ms

In C++

```cpp
NumericVector my_check_C(NumericVector x,
                         double c)
{
  // Obtain the length of vector x
  int n = x.size();

  // Allocate vector that will contain the output
  // Initialize it with x (clone makes a copy of x)
  NumericVector out = clone(x);
  int i;

  for(i = 0; i < n; i++) {
    if(x[i] <= c){
      out[i] = 0;
    }
  }
  // Vector out is returned
  return(out);
}
```

Why did we use clone()?

```
NumericVector my_check_C(NumericVector x,
                         double c)
{
  // Obtain the length of vector x
  int n = x.size();

  // Allocate vector that will contain the output
  // Initialize it with x (clone makes a copy of x)
  NumericVector out = clone(x);
  int i;

  for(i = 0; i < n; i++) {
    if(x[i] <= c){
      out[i] = 0;
    } else {
      out[i] = x[i];
    }
  }
  return(out);
}
```

Copies (clone) vs. pointers

```
NumericVector my_check_C2(NumericVector x,
                          double c)
{
  // Obtain the length of vector x
  int n = x.size();

  // Allocate vector that will contain the output
  // Now out IS x (not a copy of it)
  NumericVector out = x;
  int i;

  for(i = 0; i < n; i++) {
    if(x[i] <= c){
      out[i] = 0;
    }
  }
  return(out);
}
```

- If we don't use *clone*(), *out* IS a POINTER to *x* (not a copy of *x*)
- If we modify *out*, we also modify *x* (the original vector)
- This can be useful if we want to modify vectors in-place (in order to avoid making a copy of *x*) and save time and space

Copies (clone) vs. pointers

Without clone

```
input <- c(1,2, 3, 4, 3, 2, 1)
print(input)
my_check_C2(input, 2)
# input has changed!!
print(input)
```

```
> input <- c(1,2, 3, 4, 3, 2, 1)
> print(input) [1] 1 2 3 4 3 2 1
> my_check_C2(input, 2)
[1] 0 0 3 4 3 0 0
> # input has changed!!
> print(input)
[1] 0 0 3 4 3 0 0
```

With clone

```
input <- c(1,2, 3, 4, 3, 2, 1)
print(input)
my_check_C (input, 2)
# input has not changed
print(input)
```

```
> input <- c(1,2, 3, 4, 3, 2, 1)
> print(input)
[1] 1 2 3 4 3 2 1
> my_check_C(input, 2)
[1] 0 0 3 4 3 0 0
> # input has not changed
> print(input)
[1] 1 2 3 4 3 2 1
```

EXERCISES 2.

- Write your own my_cumsum function that computes accumulated sums, like:

```
my_cumsum(c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10))
[1]   1  3  6 10 15 21 28 36 45 55
```

```
library(Rcpp)

#####################################
# my_cumsum()
#####################################

cppFunction('
NumericVector my_cumsum(NumericVector x) {
  int n = x.size();
  # Thiscreatesa vector of length n
  NumericVector out(n);
< WRITE YOUR CODE HERE>
return out;
}
')
```

Matrix input, vector output

- R has function rowMeans, but not rowSd (!!) (sd = standard deviation)

```
> mt <- matrix(1:12, 3, 4)
> print(mt)
[,1] [,2] [,3] [,4]
[1,] 1 4 7 10
[2,] 2 5 8 11
[3,] 3 6 9 12
> rowMeans(mt)
[1] 5.5 6.5 7.5
> rowSd(mt)
Error in rowSd(mt) : could not find function "rowSd"
```

- Well, actually, library *matrixStats* contains *rowSds*(), but we are going to program *rowSd* in any case.
- Let's program rowSums_C() first, because it can be used to program *rowSd_C* in C++
- But first, let's program rowSums_R() in R

rowSums in R and in C++

```
rowSums_C(mt):                1.4 ms
rowSums_R(mt):               46.1 ms
apply(mt, 1, sum):            7.0 ms
matrixStats::rowSums2(mt): 1.8 ms
```

In R

```
rowSums_R = function(mt) {
  nrow = nrow(mt)
  ncol = ncol(mt)

  out = vector("numeric", nrow)

  for (i in 1:nrow) {
    total = 0
    for (j in 1:ncol) {
      total = total + mt[i, j]
    }
    out[i] = total
  }

  out
}
```

In C++

```
NumericVector rowSums_C(NumericMatrix mt) {
  int nrow = mt.nrow();
  int ncol = mt.ncol();

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

Matrix subsetting uses () in C++ (but [] in R)

Rcpp sugar

- Before writing the code for rowSd_C, let's introduce another concept that is going to be useful: Rcpp sugar
- In order to compute the sd:
  - First, for each row i, compute the mean
  - Then, for each row i, compute
    - sqrt[(mt(i,1) – mean(i))^2 + (mt(i,2) – mean(i))^2 + … + (mt(i,ncols) – mean(i))^2]
- We could program rowmean computation in C++, but that function already exists in R. Why not use it?
- Something called "Rcpp sugar" provides some R functions, that can be used within C++
  - List of sugar functions:
    - https://cran.r-project.org/web/packages/Rcpp/vignettes/Rcpp-sugar.pdf
    - https://github.com/coatless/rcpp-api/blob/master/rcpp-api-docs.Rmd#sugar

Matrix input, vector output. rowSD with sugar

```
NumericVector rowSd_C(NumericMatrix mt) {
  int nrow = mt.nrow();
  int ncol = mt.ncol();

  NumericVector means(nrow);
  NumericVector out(nrow);
  double total;
  int i;
  int j;

  // loop for rows
  for(i = 0; i < nrow; ++i) {
    total = 0;
   // Sugar for computing the row means
    means[i] = mean(mt(i, _));
    // loop for columns
    for (j = 0; j < ncol; ++j) {
    // pow is for squaring numbers (instead of ^2)
      total = total + pow(mt(i, j) - means[i], 2.0);
    }
    out[i] = sqrt(total/(ncol-1));
  }
  return(out);
}
```

Our function works well

```
rowSd_C(mt)
[1] 3.872983 3.872983 3.872983
> matrixStats::rowSds(mt)
[1] 3.872983 3.872983 3.872983
```

And it is faster tan matrixStats'

```
rowSd_C(mt):                2563.987 ns
matrixStats::rowSds(mt): 8259.385 ns
apply(mt, 1, sd):          96803.735 ns
```

- mt(i, _) accesses the i-th row from matrix mt
- mt(i,_) is a numeric vector

Matrix input, vector output. rowSD with more sugar

```
NumericVector rowSd_C(NumericMatrix mt) {
  int nrow = mt.nrow();
  int ncol = mt.ncol();

  NumericVector means(nrow);

  NumericVector out(nrow);
  int i;
  // More sugar: -, pow, sum are vectorized functions like in R
  for (i = 0; i < nrow; ++i) {
    means[i] = mean(mt(i, _));
    out[i] = sum(pow(mt(i, _) - means[i], 2.0));
    out[i] = sqrt(out[i]/(ncol-1));
  }
  return(out);
}
```

- mt(i, _) accesses the i-th row from matrix mt.
- mt(i,_) is a numeric vector
- mt(i,_)-means[i] is a vectorized operation
- pow(…) is a vectorized operation
- sum: takes a vector, returns the summation of all its elements

```
// loop for columns
  for (j = 0; j < ncol; ++j) {
  // pow is for squaring numbers (instead of ^2)
    total = total + pow(mt(i, j) - means[i], 2.0);
  }
```

Matrix input, vector output. rowSD with more sugar

```
NumericVector rowSd_C(NumericMatrix mt) {
  int nrow = mt.nrow();

  NumericVector out(nrow);
  int i;
  // All sugar: we just use sd
  for (i = 0; i < nrow; ++i) {
    out[i] = sd(mt(i, _));
 }
  return(out);
}
```

Matrix input, vector output

- What if we do the same loop in R

In C++

```
NumericVector rowSd_C(NumericMatrix mt) {
  int nrow = mt.nrow();

  NumericVector out(nrow);
  int i;
  // All sugar: we just use sd
  for (i = 0; i < nrow; ++i) {
    out[i] = sd(mt(i, _));
 }
  return(out);
}
```

In R

```
rowSd_R = function (mt) {
  nrow = nrow(mt)
  ncol = ncol(mt)

  means = rowMeans(mt);

  out = vector("numeric", nrow)

  for (i in 1:nrow) {
    out[i] = sd(mt[i,]);
  }

  out
}
```

Note: the same loop in R takes 8 times longer!

```
mt = matrix(runif(1000*1000), nrow = 1000)
rowSd_C(mt):                5.2 ms
matrixStats::rowSds(mt)     4.5 ms
apply(mt, 1, sd):          37.1 ms
rowSd_R(mt):               41.3 ms
```

EXERCISES 3.

- Write your own my_range function that computes the mínimum and máximum value of a vector, like:

my_range(c(3, 1, 5,2))

[1] 1 5

Use the sugar *max()* and *min()* R functions

```
library(Rcpp)

#####################################
# my_cumsum()
#####################################

cppFunction('

NumericVector my_range(NumericVector x) {
  NumericVector out(2);
  <WRITE YOUR OWN CODE HERE>
  return out;
}
')
```

R types and C++ types

	R	C++	Examples
Scalars	integer	int	3
	numeric	double	3.7
Vectors	integer	IntegerVector	(3, 5, 7)
	numeric	NumericVector	(3.8, 5.99, 7.123)
	character	CharacterVector	(“a”, “b”, “c”)
	logical	LogicalVector	(TRUE, FALSE, FALSE)
Matrices	integer	IntegerMatrix	
	numeric	NumericMatrix	
	character	CharacterMatrix	
	logical	LogicalMatrix	
Dataframes		DataFrame	
Lists		List	

Using sourceCpp and a C++ source file

- So far, we have used cppFunction and encoded the C++ function in a string
- But when the function/code is long, it is more appropriate to put all the C++ code into a standalone file
- In Rstudio, New File -> C++ file

C++ file:

We must always write this at the beginning of the file

We must always write for each function we want to export

We can also write R code that will be executed optionally

```
#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 = total + x[i];
  }
  return total;
}
```

```
// [[Rcpp::export]]
NumericVector rowSd_C(NumericMatrix mt) {
  int nrow = mt.nrow();

  // Sugar for computing the rowmeans
  NumericVector means = rowMeans(mt);

  NumericVector out(nrow);

  // More sugar: pow, sum are vectorized functions like in R
  for (int i = 0; i < nrow; i++) {
    out[i] = sd(mt(i, _));
  }
  return(out);
}
```

```
/*** R
library(microbenchmark)
x <- runif(1e5)
microbenchmark(
  sum(x),
  sumC(x)
)
*/
```

Using sourceCpp and a C++ source file

- In order to use the functions, we must compile the C++ source file with *sourceCpp('mycode.cpp')*
- It is important that the extension of the filename is ".cpp"
- We get compilation messages (with warnings and errors, sometimes):

```
C:/RBuildTools/3.4/mingw_64/bin/g++ -I"C:/PROGRA~1/R/R-35~1.1/include" -DNDEBUG -
I"C:/Users/Aler/Documents/R/win-library/3.5/Rcpp/include" -I"C:/Users/Aler/Downloads" -O2 -Wall -
mtune=generic -c mycode.cpp -o mycode.o

C:/RBuildTools/3.4/mingw_64/bin/g++ -shared -s -static-libgcc -o sourceCpp_6.dll tmp.def mycode.o -
LC:/PROGRA~1/R/R-35~1.1/bin/x64 -lR
```

Using sourceCpp and a C++ source file

- We also get the result of executing the (optional) R code

```
> library(microbenchmark)

> x <- runif(1e5)
> microbenchmark(
+   sum(x),
+   sumC(x)
+ )
Unit: microseconds
    expr    min     lq      mean median      uq     max neval
  sum(x) 68.267 68.267  68.55516 68.268 68.5110  78.264   100
 sumC(x) 91.917 92.161 100.03594 92.405 92.7705 798.721   100
```

- If execution of the R code is not desired:
  - *sourceCpp("mycode.cpp", embeddedR = FALSE)*

Using dataframes and lists

```
List changeDataframe(DataFrame DF) {

  // we receive a DF data.frame object and access each column by name
  IntegerVector a = DF["a"];
  CharacterVector b = DF["b"];
  // We use clone (copy) so that the original dataframe is not modified
  a = clone(a);
  b = clone(b);
  // do something
  a[2] = 42;
  b[1] = "foo";
  // create a new data frame with two columns named newa and newb
  DataFrame NDF = DataFrame::create(Named("newa")=a,
                                    Named("newb")=b);
  // and return old and new in list with two elements: origDataFrame and newDataFrame
  return(List::create(Named("origDataFrame")=DF,
                      Named("newDataFrame")=NDF));
}
```

Using dataframes and lists

```
> mydf <- data.frame(a=1:3,
b=letters[1:3])
> mydf
  a b
1 1 a
2 2 b
3 3 c
> mylist <-
changeDataframe(mydf)
```

```
> print(mylist)
$origDataFrame
  a b
1 1 a
2 2 b
3 3 c

$newDataFrame
  newa newb
1    1    a
2    2  foo
3   42    c
```

Extra help

- http://adv-r.had.co.nz/Rcpp.html
- https://teuder.github.io/rcpp4everyone_en/