The Rcpp Package

An Introduction

Gabriele Sarti Salvatore Milite Davide Scassola

University of Trieste
Department of Mathematics and Geosciences

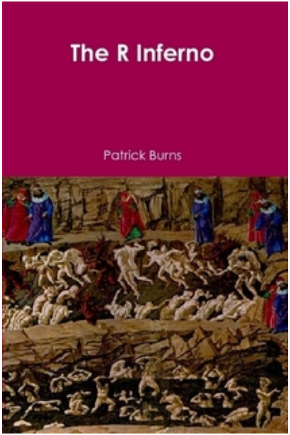
Statistical Methods for Data Science Final Exam
June 26th, 2019

Table of Contents

Table of Contents

Entering the R Inferno

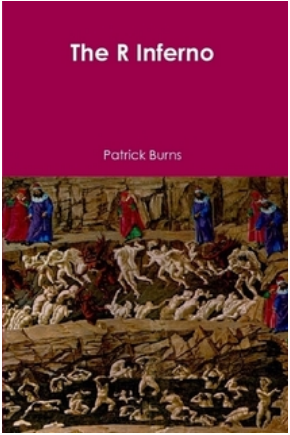
Lasciate ogne speranza, voi ch'iterate



R is both an interactive environment for *data analysis, modeling and visualization* and a language designed to support these tasks.

Entering the R Inferno

Lasciate ogne speranza, voi ch'iterate



R is both an interactive environment for *data analysis, modeling and visualization* and a language designed to support these tasks.

Most people use R to understand data, few have formal training in software development. Focus on functionality and extensibility, speed is often neglected.

Why is R Slow?

Trade-offs that limit R language performances:

- Being an extremely dynamic, interpreted language.

Why is R Slow?

Trade-offs that limit R language performances:

- Being an extremely dynamic, interpreted language.
- Name lookup with mutable environments.

Why is R Slow?

Trade-offs that limit R language performances:

- Being an extremely dynamic, interpreted language.
- Name lookup with mutable environments.
- Overhead coming from lazy evaluation of function arguments.

Why is R Slow?

Trade-offs that limit R language performances:

- Being an extremely dynamic, interpreted language.
- Name lookup with mutable environments.
- Overhead coming from lazy evaluation of function arguments.

Additional limitations imposed by the implementation:

- Lack of efficient built-in types.

Why is R Slow?

Trade-offs that limit R language performances:

- Being an extremely dynamic, interpreted language.
- Name lookup with mutable environments.
- Overhead coming from lazy evaluation of function arguments.

Additional limitations imposed by the implementation:

- Lack of efficient built-in types.
- Functions that are too general to be fast.

Why is R Slow?

Trade-offs that limit `R` language performances:

- Being an extremely dynamic, interpreted language.
- Name lookup with mutable environments.
- Overhead coming from lazy evaluation of function arguments.

Additional limitations imposed by the implementation:

- Lack of efficient built-in types.
- Functions that are too general to be fast.
- `R-core` is bad performance-wise and won't be changed.

Why is R Slow?

Trade-offs that limit `R` language performances:

- Being an extremely dynamic, interpreted language.
- Name lookup with mutable environments.
- Overhead coming from lazy evaluation of function arguments.

Additional limitations imposed by the implementation:

- Lack of efficient built-in types.
- Functions that are too general to be fast.
- `R-core` is bad performance-wise and won't be changed.

Vectorization may help, *if you do it properly*.

Using the C Interface

The core interpreter and the extension mechanism of `R` are both implemented using `C`. Thus, the two languages are compatible.

Using the C Interface

The core interpreter and the extension mechanism of `R` are both implemented using `C`. Thus, the two languages are compatible.

A possible approach to improve performance:

- Create a `C` file with `R` headers to use `SEXP` data type.

Using the C Interface

The core interpreter and the extension mechanism of `R` are both implemented using `C`. Thus, the two languages are compatible.

A possible approach to improve performance:

- Create a `C` file with `R` headers to use `SEXP` data type.
- Write the function in `C` language.

Using the C Interface

The core interpreter and the extension mechanism of `R` are both implemented using `C`. Thus, the two languages are compatible.

A possible approach to improve performance:

- Create a `C` file with `R` headers to use `SEXP` data type.
- Write the function in `C` language.
- Compile the file using `R CMD` in command line.

Using the C Interface

The core interpreter and the extension mechanism of `R` are both implemented using `C`. Thus, the two languages are compatible.

A possible approach to improve performance:

- Create a `C` file with `R` headers to use `SEXP` data type.
- Write the function in `C` language.
- Compile the file using `R CMD` in command line.
- Call the executable from `R` using `dyn.load(file.o).Call("function", param)`

Using the C Interface

The core interpreter and the extension mechanism of `R` are both implemented using `C`. Thus, the two languages are compatible.

A possible approach to improve performance:

- Create a `C` file with `R` headers to use `SEXP` data type.
- Write the function in `C` language.
- Compile the file using `R CMD` in command line.
- Call the executable from `R` using `dyn.load(file.o).Call("function", param)`

Long, error-prone and we still cannot use classes and the `STL`!

Table of Contents

The Rcpp Package

The Rcpp package is currently the most widely used language extension for R, with over 1000 CRAN packages using it to accelerate computations and to connect to other C++ projects.

The Rcpp Package

The `Rcpp` package is currently the most widely used language extension for `R`, with over 1000 `CRAN` packages using it to accelerate computations and to connect to other `C++` projects.

`Rcpp` provides a clean programming interface to encapsulate `R`'s complex `C API` and write high-performance code with small effort.

The Rcpp Package

The Rcpp package is currently the most widely used language extension for R, with over 1000 CRAN packages using it to accelerate computations and to connect to other C++ projects.

Rcpp provides a clean programming interface to encapsulate R's complex C API and write high-performance code with small effort.

Typical issues addressed by Rcpp:

- Loops with inter-iteration dependencies.

The Rcpp Package

The Rcpp package is currently the most widely used language extension for R, with over 1000 CRAN packages using it to accelerate computations and to connect to other C++ projects.

Rcpp provides a clean programming interface to encapsulate R's complex C API and write high-performance code with small effort.

Typical issues addressed by Rcpp:

- Loops with inter-iteration dependencies.
- Recursive functions.

The Rcpp Package

The `Rcpp` package is currently the most widely used language extension for `R`, with over 1000 `CRAN` packages using it to accelerate computations and to connect to other `C++` projects.

`Rcpp` provides a clean programming interface to encapsulate `R`'s complex `C API` and write high-performance code with small effort.

Typical issues addressed by `Rcpp`:

- Loops with inter-iteration dependencies.
- Recursive functions.
- Problems requiring optimized algorithms and data structures.

The Rcpp Package

The `Rcpp` package is currently the most widely used language extension for `R`, with over 1000 `CRAN` packages using it to accelerate computations and to connect to other `C++` projects.

`Rcpp` provides a clean programming interface to encapsulate `R`'s complex `C API` and write high-performance code with small effort.

Typical issues addressed by `Rcpp`:

- Loops with inter-iteration dependencies.
- Recursive functions.
- Problems requiring optimized algorithms and data structures.

A Simple Example: The Old Way

```
#include <R.h>
#include <Rdefines.h>

int fibonacci(const int x) {
  if (x == 0) return(0);
  if (x == 1) return(1);
  return fibonacci(x - 1) + fibonacci(x - 2);
}

extern "C" SEXP fibWrapper(SEXP xs) {
  int x = Rcpp::as<int>(xs);
  int fib = fibonacci(x);
  return (Rcpp::wrap(fib));
}
```

↓

```
[gsarti@antergos ~]$ R CMD SHLIB fibo.c
```

↓

```
> dyn.load("fibo.so")
    .Call("fibonacci", as.integer(5))
  [1] 3
```

A Simple Example: The Old Way

```c
#include <R.h>
#include <Rdefines.h>

int fibonacci(const int x) {
  if (x == 0) return(0);
  if (x == 1) return(1);
  return fibonacci(x - 1) + fibonacci(x - 2);
}

extern "C" SEXP fibWrapper(SEXP xs) {
  int x = Rcpp::as<int>(xs);
  int fib = fibonacci(x);
  return (Rcpp::wrap(fib));
}
```

↓

```
[gsarti@antergos ~]$ R CMD SHLIB fibo.c
```

↓

```r
> dyn.load("fibo.so")
    .Call("fibonacci", as.integer(5))
  [1] 3
```

```r
## Using a pure C++ function to allow recursive
## calls thanks to C++ identifier.
txt <- '
int fibonacci(const int x) {
  if (x == 0) return(0);
  if (x == 1) return(1);
  return fibonacci(x - 1) + fibonacci(x - 2);
}'

fiboRcpp <- cxxfunction(signature(xs="int"),
                        plugin="Rcpp",
                        incl=txt,
                        body='
         int x = Rcpp::as<int>(xs);
         return Rcpp::wrap( fibonacci(x) );')
```

A Simple Example: The Old Way

```c
#include <R.h>
#include <Rdefines.h>

int fibonacci(const int x) {
  if (x == 0) return(0);
  if (x == 1) return(1);
  return fibonacci(x - 1) + fibonacci(x - 2);
}

extern "C" SEXP fibWrapper(SEXP xs) {
  int x = Rcpp::as<int>(xs);
  int fib = fibonacci(x);
  return (Rcpp::wrap(fib));
}
```

↓

```
[gsarti@antergos ~]$ R CMD SHLIB fibo.c
```

↓

```r
> dyn.load("fibo.so")
   .Call("fibonacci", as.integer(5))
  [1] 3
```

```r
## Using a pure C++ function to allow recursive
## calls thanks to C++ identifier.
txt <- '
int fibonacci(const int x) {
  if (x == 0) return(0);
  if (x == 1) return(1);
  return fibonacci(x - 1) + fibonacci(x - 2);
}'

fiboRcpp <- cxxfunction(signature(xs="int"),
                        plugin="Rcpp",
                        incl=txt,
                        body='
         int x = Rcpp::as<int>(xs);
         return Rcpp::wrap( fibonacci(x) );')
```

Still using `as` an `wrap` functions from the original `C API` through the `inline` package, still not intuitive enough.

A Simple Example: The Modern Way

```
// Inside fibo.cpp
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
int fibonacci1(const int x) {
  if (x == 0) return(0);
  if (x == 1) return(1);
  return fibonacci(x - 1) + fibonacci(x - 2);
}

// [[Rcpp::export]]
int fibonacci2(const int x) {
  if (x < 2) return x;
  return fibonacci(x - 1) + fibonacci(x - 2);
}

// Inside the R file calling fibonacci
sourceCpp("fibo.cpp")
fibonacci1(20)
fibonacci2(20)
```

A Simple Example: The Modern Way

```cpp
// Inside fibo.cpp
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
int fibonacci1(const int x) {
  if (x == 0) return(0);
  if (x == 1) return(1);
  return fibonacci(x - 1) + fibonacci(x - 2);
}

// [[Rcpp::export]]
int fibonacci2(const int x) {
  if (x < 2) return x;
  return fibonacci(x - 1) + fibonacci(x - 2);
}

// Inside the R file calling fibonacci
sourceCpp("fibo.cpp")
fibonacci1(20)
fibonacci2(20)
```

```r
cppFunction(
'int fibonacci(const int x) {
    if (x == 0) return(0);
    if (x == 1) return(1);
    return fibonacci(x - 1) + fibonacci(x - 2);
 }')
> fibonacci(20)
[1] 6765
```

A Simple Example: The Modern Way

```
// Inside fibo.cpp
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
int fibonacci1(const int x) {
  if (x == 0) return(0);
  if (x == 1) return(1);
  return fibonacci(x - 1) + fibonacci(x - 2);
}

// [[Rcpp::export]]
int fibonacci2(const int x) {
  if (x < 2) return x;
  return fibonacci(x - 1) + fibonacci(x - 2);
}

// Inside the R file calling fibonacci
sourceCpp("fibo.cpp")
fibonacci1(20)
fibonacci2(20)
```

```
cppFunction(
'int fibonacci(const int x) {
    if (x == 0) return(0);
    if (x == 1) return(1);
    return fibonacci(x - 1) + fibonacci(x - 2);
 }')
```

```
> fibonacci(20)
[1] 6765
```

Run-time performance of the recursive Fibonacci examples

Function	N	Elapsed time (s)	Relative (ratio)
fibRcpp	1	0.092	1.00
fibR	1	62.288	677.04
(byte-compiled) fibR	1	62.711	681.64

A Simple Example: The Modern Way

```cpp
// Inside fibo.cpp
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
int fibonacci1(const int x) {
  if (x == 0) return(0);
  if (x == 1) return(1);
  return fibonacci(x - 1) + fibonacci(x - 2);
}

// [[Rcpp::export]]
int fibonacci2(const int x) {
  if (x < 2) return x;
  return fibonacci(x - 1) + fibonacci(x - 2);
}

// Inside the R file calling fibonacci
sourceCpp("fibo.cpp")
fibonacci1(20)
fibonacci2(20)
```

```r
cppFunction(
'int fibonacci(const int x) {
    if (x == 0) return(0);
    if (x == 1) return(1);
    return fibonacci(x - 1) + fibonacci(x - 2);
 }')
> fibonacci(20)
[1] 6765
```

Run-time performance of the recursive Fibonacci examples

Function	N	Elapsed time (s)	Relative (ratio)
fibRcpp	1	0.092	1.00
fibR	1	62.288	677.04
(byte-compiled) fibR	1	62.711	681.64

Behind the Scenes of Rcpp Attributes

`C++11` attributes referred by `[[Rcpp::export]]` encapsulated the complexity of the `inline` package.

Behind the Scenes of Rcpp Attributes

C++11 attributes referred by `[[Rcpp::export]]` encapsulated the complexity of the `inline` package.

The new functions `cppFunction`, `sourceCpp` and `evalCpp` are thus much simpler. Using `/*** R` inside C++ files is also allowed!

Behind the Scenes of Rcpp Attributes

C++11 attributes referred by [[Rcpp::export]] encapsulated the complexity of the inline package.

The new functions cppFunction, sourceCpp and evalCpp are thus much simpler. Using /*** R inside C++ files is also allowed!

- cppFunction returns an R function which calls a wrapper in a temporary file which it also builds, which in turn calls the C++ function we passed as a character string.

Behind the Scenes of Rcpp Attributes

C++11 attributes referred by [[Rcpp::export]] encapsulated the complexity of the inline package.

The new functions cppFunction, sourceCpp and evalCpp are thus much simpler. Using /*** R inside C++ files is also allowed!

- cppFunction returns an R function which calls a wrapper in a temporary file which it also builds, which in turn calls the C++ function we passed as a character string.
- sourceCpp compiles, links, and loads the corresponding C++ source file.

Behind the Scenes of Rcpp Attributes

C++11 attributes referred by `[[Rcpp::export]]` encapsulated the complexity of the `inline` package.

The new functions `cppFunction`, `sourceCpp` and `evalCpp` are thus much simpler. Using `/*** R` inside C++ files is also allowed!

- `cppFunction` returns an R function which calls a wrapper in a temporary file which it also builds, which in turn calls the C++ function we passed as a character string.
- `sourceCpp` compiles, links, and loads the corresponding C++ source file.

A caching mechanism ensures a single compilation of the code.

Table of Contents

SEXP and SEXPREC

Objects in `R` are internally represented as `SEXP`, a pointer to a so-called S-expression object, or `SEXPREC` for short (or `VECSXP` and `VECSXPREC` for vectors).

SEXP and SEXPREC

Objects in `R` are internally represented as `SEXP`, a pointer to a so-called S-expression object, or `SEXPREC` for short (or `VECSXP` and `VECSXPREC` for vectors).

`SEXPR` objects are union types (which are sometimes called variant types). This means that depending on the particular value in an 64 bit header, different types can be represented.

SEXP and SEXPREC

Objects in `R` are internally represented as `SEXP`, a pointer to a so-called S-expression object, or `SEXPREC` for short (or `VECSXP` and `VECSXPREC` for vectors).

`SEXPR` objects are union types (which are sometimes called variant types). This means that depending on the particular value in an 64 bit header, different types can be represented.

`SEXP` objects should be considered opaque, i.e. should be accessed only by macros provided by the `R API`. In this sense `Rcpp API` provides an higher level of abstraction

The RObject Class

The `RObject` class is the basic class underlying the `Rcpp API`. It is the parent of all the classes we are going to see next and it is the main ingredient of the package abstraction.

The RObject Class

The `RObject` class is the basic class underlying the `Rcpp API`. It is the parent of all the classes we are going to see next and it is the main ingredient of the package abstraction.

We can think of an `RObject` as wrapper around the `SEXP` structure (In fact, the `SEXP` is indeed the only data member of an `RObject`)

The RObject Class

The `RObject` class is the basic class underlying the `Rcpp API`. It is the parent of all the classes we are going to see next and it is the main ingredient of the package abstraction.

We can think of an `RObject` as wrapper around the `SEXP` structure (In fact, the `SEXP` is indeed the only data member of an `RObject`)

The main idea is that all the functions that directly access the `SEXP` object are implemented in this class, this gives the user a much more transparent way to interact with `R` internals.

Numeric Vector and Integer Vector

Is pretty rare to instantiate a raw `RObject`. It is preferable to work with the derived classes which are specific for each `R` type.

Numeric Vector and Integer Vector

Is pretty rare to instantiate a raw `RObject`. It is preferable to work with the derived classes which are specific for each R type.

Templated function `as<>()` and class constructors with R objects as argument are used to convert a `SEXP` structure to the corresponding `Rcpp` type. The `SEXP` to R can be returned by using `wrap()`.

Numeric Vector and Integer Vector

Is pretty rare to instantiate a raw `RObject`. It is preferable to work with the derived classes which are specific for each `R` type.

Templated function `as<>()` and class constructors with `R` objects as argument are used to convert a `SEXP` structure to the corresponding `Rcpp` type. The `SEXP` to `R` can be returned by using `wrap()`.

Two of the most commonly used `Rcpp` classes are:

- Numeric Vector: corresponds to the basic `R` type of a numeric vector and can hold real-valued floating-point variables.

Numeric Vector and Integer Vector

Is pretty rare to instantiate a raw `RObject`. It is preferable to work with the derived classes which are specific for each `R` type.

Templated function `as<>()` and class constructors with `R` objects as argument are used to convert a `SEXP` structure to the corresponding `Rcpp` type. The `SEXP` to `R` can be returned by using `wrap()`.

Two of the most commonly used `Rcpp` classes are:

- Numeric Vector: corresponds to the basic `R` type of a numeric vector and can hold real-valued floating-point variables.
- Integer Vector: Provides a natural mapping from and to the standard `R` integer vectors.

Other Vector Classes

Other common used vector classes are:

- Logical Vector for storing boolean values.
- Character Vector for strings.
- Raw Vector for raw bytes.

Other Vector Classes

Other common used vector classes are:

- Logical Vector for storing boolean values.
- Character Vector for strings.
- Raw Vector for raw bytes.

Numeric and Integer matrix are also present. Vectors are actually implemented as multidimensional, their dimension can be changed by providing an attribute in the object constructor.

Other Vector Classes

Other common used vector classes are:

- Logical Vector for storing boolean values.
- Character Vector for strings.
- Raw Vector for raw bytes.

Numeric and Integer matrix are also present. Vectors are actually implemented as multidimensional, their dimension can be changed by providing an attribute in the object constructor.

Another useful feature of the vector family is the the presence of already implemented iterators, which gives full compatibility with the `C++` STL methods.

Named and List

The Named class is an helper class used for setting the key side of key/value pairs. It corresponds to R's `c()` map construct.

```
Rcpp::NumericVector x =
    Rcpp::NumericVector::create(
        Rcpp::Named("mean") = 1.23,
        Rcpp::Named("dim") = 42,
        Rcpp::Named("cnt") = 12);
```

Named and List

The Named class is an helper class used for setting the key side of key/value pairs. It corresponds to R's `c()` map construct.

```
Rcpp::NumericVector x =
    Rcpp::NumericVector::create(
        Rcpp::Named("mean") = 1.23,
        Rcpp::Named("dim") = 42,
        Rcpp::Named("cnt") = 12);
```

The Generic Vector class is the equivalent of the `List` type in R. It can contain objects of different types, including other generic vectors. Because of its flexibility it is commonly used to parameter exchanges in either directions.

Function and Dataframe

The Function class helps us whenever we want to use an `R` function in `Rcpp`. It is possible to either pass the `Function` object or to retrieve it from the environment by name.

Function and Dataframe

The Function class helps us whenever we want to use an `R` function in `Rcpp`. It is possible to either pass the `Function` object or to retrieve it from the environment by name.

The Dataframe class is, much like its `R` counterpart, implemented as lists constrained to have elements of the same length. However, while `R` dataframes can *recycle*, i.e. elements of the shorter columns can be repeated until a valid `Dataframe` structure is obtained, this is not possible in `Rcpp` dataframes and will raise an exception.

Environments

In Rcpp the Environment class can be used to retrieve and modify elements in an R environment.

Environments

In Rcpp the Environment class can be used to retrieve and modify elements in an R environment.

Environments associate a set of names to a set of values. They can be thought as named lists but with some differences:

- Every name in an environment is unique.

Environments

In Rcpp the Environment class can be used to retrieve and modify elements in an R environment.

Environments associate a set of names to a set of values. They can be thought as named lists but with some differences:

- Every name in an environment is unique.
- The names in an environment are not ordered.

Environments

In Rcpp the Environment class can be used to retrieve and modify elements in an R environment.

Environments associate a set of names to a set of values. They can be thought as named lists but with some differences:

- Every name in an environment is unique.
- The names in an environment are not ordered.
- An environment has always a parent (except the empty environment).

Environments

In Rcpp the Environment class can be used to retrieve and modify elements in an R environment.

Environments associate a set of names to a set of values. They can be thought as named lists but with some differences:

- Every name in an environment is unique.
- The names in an environment are not ordered.
- An environment has always a parent (except the empty environment).
- Environments have reference semantics.

Environments

In Rcpp the Environment class can be used to retrieve and modify elements in an R environment.

Environments associate a set of names to a set of values. They can be thought as named lists but with some differences:

- Every name in an environment is unique.
- The names in an environment are not ordered.
- An environment has always a parent (except the empty environment).
- Environments have reference semantics.

Namespaces are implemented using environments, and environment enable the use of *closures* as the standard for functions in R.

The R Object Oriented System

There are three main object systems in `R`:

- S3 implements *generic-function object orientation*, different from the more common message-passing object orientation where the methods belongs to a class. Computations are still performed via methods, but a generic function decides which method to call. `S3` has no formal definition of classes.

The R Object Oriented System

There are three main object systems in R:

- S3 implements *generic-function object orientation*, different from the more common message-passing object orientation where the methods belongs to a class. Computations are still performed via methods, but a generic function decides which method to call. S3 has no formal definition of classes.
- S4 are just S3 objects with more formalism. They have a rigorous definition of attributes and inheritance, and have helper functions for defining generics and methods.

The R Object Oriented System

There are three main object systems in R:

- S3 implements *generic-function object orientation*, different from the more common message-passing object orientation where the methods belongs to a class. Computations are still performed via methods, but a generic function decides which method to call. S3 has no formal definition of classes.
- S4 are just S3 objects with more formalism. They have a rigorous definition of attributes and inheritance, and have helper functions for defining generics and methods.
- RC implements message-passing object orientation. RC objects are also mutable: they dont use Rs usual copy-on-modify semantics, but are modified in place.

S4 and RC

The `Rcpp S4` class allows access and modification of `S4` objects attributes, allowing to test if an `RObject` is a `S4`. Nevertheless, it is a good practice to manipulate the object and isolate the interesting attributes directly in `R`, and then carry out the computations in `Rcpp` using simpler types.

The `Rcpp RC` class provides a natural interface for `R RC` classes. It is mostly used for mutable data-structures, particularly those that deal with graphics and stream of data. However, `RC` is not a common data-type and it is not firmly established in the community.

Table of Contents

Rcpp Sugar

Rcpp provides a lot of syntactic sugar expressions to ensure that C++ functions work very similarly to their R equivalents. Rcpp sugar makes possible to write efficient C++ code that looks almost identical to its R equivalent.

- arithmetic and logical vectorized operators: $+, *, -, /$, pow, $<, <=, >, >=, ==, !=, !$

Rcpp Sugar

Rcpp provides a lot of syntactic sugar expressions to ensure that C++ functions work very similarly to their R equivalents. Rcpp sugar makes possible to write efficient C++ code that looks almost identical to its R equivalent.

- arithmetic and logical vectorized operators: $+, *, -, /$, pow, $<, <=, >, >=, ==, !=, !$
- logical summary functions: `any()`, `all()`

Rcpp Sugar

`Rcpp` provides a lot of syntactic sugar expressions to ensure that `C++` functions work very similarly to their `R` equivalents. `Rcpp` sugar makes possible to write efficient `C++` code that looks almost identical to its R equivalent.

- arithmetic and logical vectorized operators: $+, *, -, /$, $\texttt{pow}, <, <=, >, >=, ==, != , !$
- logical summary functions: `any()`, `all()`
- scalar summaries: `mean()`, `min()`, `max()`, `sum()`, `sd()`, ...

Rcpp Sugar

Rcpp provides a lot of syntactic sugar expressions to ensure that C++ functions work very similarly to their R equivalents. Rcpp sugar makes possible to write efficient C++ code that looks almost identical to its R equivalent.

- arithmetic and logical vectorized operators: $+, *, -, /$, pow, $<, <=, >, >=, ==, !=, !$
- logical summary functions: `any()`, `all()`
- scalar summaries: `mean()`, `min()`, `max()`, `sum()`, `sd()`, ...
- vector views: `head()`, `tail()`, `rep_len()`, ...

Rcpp Sugar

`Rcpp` provides a lot of syntactic sugar expressions to ensure that `C++` functions work very similarly to their `R` equivalents. `Rcpp` sugar makes possible to write efficient `C++` code that looks almost identical to its R equivalent.

- arithmetic and logical vectorized operators: $+, *, -, /$, `pow`, $<, <=, >, >=, ==, !=, !$
- logical summary functions: `any()`, `all()`
- scalar summaries: `mean()`, `min()`, `max()`, `sum()`, `sd()`, ...
- vector views: `head()`, `tail()`, `rep_len()`, ...
- basic math functions: `abs()`, `log()`, `sin()`, ...

Rcpp Sugar

Rcpp provides a lot of syntactic sugar expressions to ensure that C++ functions work very similarly to their R equivalents. Rcpp sugar makes possible to write efficient C++ code that looks almost identical to its R equivalent.

- arithmetic and logical vectorized operators: $+, *, -, /,$ pow$, <, <=, >, >=, ==, !=, !$
- logical summary functions: `any()`, `all()`
- scalar summaries: `mean()`, `min()`, `max()`, `sum()`, `sd()`, ...
- vector views: `head()`, `tail()`, `rep_len()`, ...
- basic math functions: `abs()`, `log()`, `sin()`, ...
- distributions: `runif()`, `dnorm()`, ...

Rcpp Sugar: An Example

```r
piR <-function(N)
{
  x <-runif(N)
  y <-runif(N)
  d <-sqrt(x^2+y^2)
  return(4*sum(d <=1.0) /N)
}
```

Rcpp Sugar: An Example

```r
piR <-function(N)
{
  x <-runif(N)
  y <-runif(N)
  d <-sqrt(x^2+y^2)
  return(4*sum(d <=1.0) /N)
}
```

```cpp
double piSugar(const int N)
{
  NumericVector x = runif(N);
  NumericVector y = runif(N);
  NumericVector d = sqrt(x*x + y*y);
  return 4.0* sum(d <=1.0) / N;
}
```

Rcpp Sugar: An Example

```r
piR <-function(N)
{
  x <-runif(N)
  y <-runif(N)
  d <-sqrt(x^2+y^2)
  return(4*sum(d <=1.0) /N)
}
```

```cpp
double piSugar(const int N)
{
  NumericVector x = runif(N);
  NumericVector y = runif(N);
  NumericVector d = sqrt(x*x + y*y);
  return 4.0* sum(d <=1.0) / N;
}
```

The only difference is the type declaration and the missing operator ˆ in C++.

Rcpp Sugar: Performances

Table 8.1 Run-time performance of *Rcpp sugar* compared to R and manually optimized C++

R expression	Runs	Manual	Sugar	R
`any(x * y < 0)`	5,000	0.00027	0.00069	6.8914
`ifelse(x<y, x*x, -(y*y))`	500	1.28566	1.52103	13.8829
`ifelse(x<y, x*x, -(y*y))` (noNA)	500	0.41462	1.14434	13.8537
`sapply(x, square)`	500	0.16721	0.19224	115.4236

Sugar functions don't always perform like hand-written ones in `C++` (for now!)

Exposing C++ Classes to R

We've already seen how to use `C++` functions in `R`, now we want to use an entire user-defined `C++` class in `R`.

- Unlike functions, it does not exist an attribute shortcut for classes

Exposing C++ Classes to R

We've already seen how to use `C++` functions in `R`, now we want to use an entire user-defined `C++` class in `R`.

- Unlike functions, it does not exist an attribute shortcut for classes
- Anyway, it's possible to avoid writing tedious operational code. How?

Exposing C++ Classes to R

We've already seen how to use `C++` functions in `R`, now we want to use an entire user-defined `C++` class in `R`.

- Unlike functions, it does not exist an attribute shortcut for classes
- Anyway, it's possible to avoid writing tedious operational code. How?
- Thanks to the macro `RCPP_MODULE`, that `Rcpp` provides.

Exposing C++ Classes to R

We've already seen how to use `C++` functions in `R`, now we want to use an entire user-defined `C++` class in `R`.

- Unlike functions, it does not exist an attribute shortcut for classes
- Anyway, it's possible to avoid writing tedious operational code. How?
- Thanks to the macro `RCPP_MODULE`, that `Rcpp` provides.
- It can be used also for exposing functions.

Exposing C++ Classes: An Example

```cpp
using namespace Rcpp;
class Uniform {
public:
    Uniform(double min_, double max_) : min(min_), max(max_) {}

    NumericVector draw(int n) const {
        RNGScope scope;
        return runif( n, min, max );
    }

    double min, max;
};

double uniformRange( Uniform* w) {
    return w->max - w->min;
}
```

Exposing C++ classes: An Example

We just have to choose a name for the module, and declare a name for every class member.

```
RCPP_MODULE(unif_module) {

    class_<Uniform>( "Uniform" )

    .constructor<double,double>()

    .field( "min", &Uniform::min )
    .field( "max", &Uniform::max )

    .method( "draw", &Uniform::draw )
    .method( "range", &uniformRange )
    ;

}
```

Exposing C++ classes: An Example

After compiling the .cpp file as a dynamic library, we load the module in R, and now it's ready to be used like a class.

```
library(Rcpp)
library(inline)

setwd("/home/davide/Scrivania/StatisticalMethods/Rcpp/boot_test")

lib <- dyn.load("module.so")
unif_module <- Module("unif_module", lib)
Uniform <- unif_module$Uniform
```

Now we can use the Uniform class:

```
u <- new(Uniform, 0, 10)
u$range()
```

```
## [1] 10
```

```
u$draw(10)
```

```
##  [1] 6.782685 7.203999 8.192901 5.990833 1.390944 6.746467 7.018502
##  [8] 7.937970 8.573115 1.653107
```

Extending Rcpp

Many packages are available to extend base `Rcpp` functionalities:

- `Rinside`: Embedded `R` in `C++` applications.

Extending Rcpp

Many packages are available to extend base `Rcpp` functionalities:

- `Rinside`: Embedded `R` in `C++` applications.
- `RcppArmadillo`: Interface to `Armadillo`, a linear algebra library that balances speed and ease of use.

Extending Rcpp

Many packages are available to extend base `Rcpp` functionalities:

- `Rinside`: Embedded `R` in `C++` applications.
- `RcppArmadillo`: Interface to `Armadillo`, a linear algebra library that balances speed and ease of use.
- `RcppEigen`: Interface to the `Eigen` library, a library suited for dealing with matrix decomposition and sparse matrices.

Extending Rcpp

Many packages are available to extend base `Rcpp` functionalities:

- `Rinside`: Embedded `R` in `C++` applications.
- `RcppArmadillo`: Interface to `Armadillo`, a linear algebra library that balances speed and ease of use.
- `RcppEigen`: Interface to the `Eigen` library, a library suited for dealing with matrix decomposition and sparse matrices.

RcppArmadillo: An Example

```cpp
#include<RcppArmadillo.h>

// [[Rcpp::depends(RcppArmadillo)]]

//[[Rcpp::export]]
arma::vec getEigenValues(arma::mat M)
  {
  return arma::eig_sym(M);
  }
```

RcppArmadillo: An Example

```cpp
#include<RcppArmadillo.h>

// [[Rcpp::depends(RcppArmadillo)]]

//[[Rcpp::export]]
arma::vec getEigenValues(arma::mat M)
  {
  return arma::eig_sym(M);
  }
```

```r
> sourceCpp("armadilloExample.cpp")
> X <-matrix(rnorm(4*4), 4, 4)
> Z <- X %*%t(X)
> getEigenValues(Z)
          [,1]
[1,] 0.1033663
[2,] 1.5578299
[3,] 3.0959708
[4,] 8.4696693
```

RcppArmadillo: An Example

```cpp
#include<RcppArmadillo.h>

// [[Rcpp::depends(RcppArmadillo)]]

//[[Rcpp::export]]
arma::vec getEigenValues(arma::mat M)
  {
  return arma::eig_sym(M);
  }
```

```r
> sourceCpp("armadilloExample.cpp")
> X <-matrix(rnorm(4*4), 4, 4)
> Z <- X %*%t(X)
> getEigenValues(Z)
          [,1]
[1,] 0.1033663
[2,] 1.5578299
[3,] 3.0959708
[4,] 8.4696693
```

Notice that using attributes it is possible to declare dependencies on other packages.

Table of Contents

Bootstrap

Definition

Bootstrapping is a type of resampling where large numbers of smaller samples of the same size are repeatedly drawn, with replacement, from a single original sample.

In order to show a possible use case of `Rcpp` in statistics, we will write a function that computes the means and the standard deviations for a set of B samples bootstrapped from an original data set of size n.

Since the process is iterative by definition, we expect a `for` loop in the implementation, which can possibly be optimized by `Rcpp`.

Bootstrap, The R way

```r
bootstrap_r <- function(ds, B = 1000)
 {
  # Preallocate storage for statistics
  boot_stat <- matrix(NA, nrow = B, ncol = 2)

  n <- length(ds)

  # Perform bootstrap
  for(i in seq_len(B))
  {
    # Sample initial data
    gen_data <- ds[ sample(n, n, replace=TRUE) ]

    # Calculate sample data mean and SD
    boot_stat[i,] <- c(mean(gen_data), sd(gen_data))
  }

  # Return bootstrap result
  return(boot_stat)
}
```

Bootstrap, The C++ way

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

// [[Rcpp::export]]
Rcpp::NumericMatrix bootstrap_cpp(Rcpp::NumericVector ds, const int B = 1000)
{
  // Preallocate storage for statistics
  Rcpp::NumericMatrix boot_stat(B, 2);

  int n = ds.size();

  // Perform bootstrap
  for(int i = 0; i < B; i++)
  {
    // Sample initial data
    Rcpp::NumericVector gen_data = ds[ floor(Rcpp::runif(n, 0, n)) ];

    boot_stat(i, 0) = mean(gen_data); // sample mean
    boot_stat(i, 1) = sd(gen_data); // sample std dev
  }

  // Return bootstrap results
  return boot_stat;
}
```

Bootstrap Performances

```r
library(Rcpp)
library(rbenchmark)
library(inline)

B <- 10^4
N <- 100
data <- rnorm(n = N, 2, 4) + rnorm(n = N,3,1)

sourceCpp("bootCpp.cpp")
benchmark(r = bootstrap_r(data,B), cpp = bootstrap_cpp(data, B), replications = 10)
```

```
> benchmark(r = bootstrap_r(data,B), cpp = bootstrap_cpp(data, B), replications = 10)
  test replications elapsed relative user.self sys.self user.child sys.child
2  cpp           10   0.305    1.000     0.305        0          0         0
1    r           10   3.032    9.941     3.033        0          0         0
> 
```

Our Rcpp approach is almost ten times faster!

Table of Contents

References I

Douglas Bates, Dirk Eddelbuettel, et al., *Fast and elegant numerical linear algebra using the rcppeigen package*, Journal of Statistical Software 52 (2013), no. 5, 1–24.

Dirk Eddelbuettel and James Joseph Balamuta, *Extending r with c++: A brief introduction to rcpp*, The American Statistician 72 (2018), no. 1, 28–36.

Dirk Eddelbuettel, *Rcpp Documentation*, `http://dirk.eddelbuettel.com/code/rcpp/html/index.html`, [Online; accessed 22 June 2019].

______, *Rinside*, Seamless R and C++ Integration with Rcpp, Springer, 2013, pp. 127–137.

References II

- ______, *Seamless r and c++ integration with rcpp*, Springer, 2013.
- Dirk Eddelbuettel, Romain François, J Allaire, Kevin Ushey, Qiang Kou, N Russel, John Chambers, and D Bates, *Rcpp: Seamless r and c++ integration*, Journal of Statistical Software 40 (2011), no. 8, 1–18.
- Dirk Eddelbuettel and Conrad Sanderson, *Rcpparmadillo: Accelerating r with high-performance c++ linear algebra*, Computational Statistics & Data Analysis 71 (2014), 1054–1063.

References III

- Floréal Morandat, Brandon Hill, Leo Osvald, and Jan Vitek, *Evaluating the design of the r language*, European Conference on Object-Oriented Programming, Springer, 2012, pp. 104–131.
- Hadley Wickham, *Advanced r*, Chapman and Hall/CRC, 2014.