Higher-Performance R via C++

Part 1: Introduction

Dirk Eddelbuettel
UZH/ETH Zürich R Courses
June 24-25, 2015
Overview
High-level motivation: Three main questions

- Why? *Several reasons discussed next*
- How? *Rcpp details, usage, tips, ...*
- What? *We will cover examples.*
Focus on R and C++

- R: Our starting point
- C++: Our extension approach
- why, how, tricks, ...
Maybe some mutual introductions?

- Your background (academic, industry, ...)
- R experience (beginner, intermediate, advanced, ...)
- Created / modified any R packages?
- C and/or C++ experience?
- Main interest in Rcpp: speed, extensions, ..., ?
- Following rcpp-devel or r-devel?
Overview: Why R?
Why R? : Programming with Data


Thanks to John Chambers for sending me high-resolution scans of the covers of his books.
xx <- faithful[, "eruptions"]
fit <- density(xx)
plot(fit)
A Simple Example

density.default(x = xx)

N = 272   Bandwidth = 0.3348
xx <- faithful[, "eruptions"]
fit1 <- density(xx)
fit2 <- replicate(10000, {
    x <- sample(xx, replace=TRUE);
    density(x, from=min(fit1$x), to=max(fit1$x))$y
})
fit3 <- apply(fit2, 1, quantile, c(0.025, 0.975))
plot(fit1, ylim=range(fit3))
polygon(c(fit1$x, rev(fit1$x)), c(fit3[1,], rev(fit3[2,])),
        col='grey', border=F)
lines(fit1)
A Simple Example - Refined

density.default(x = xx)

N = 272   Bandwidth = 0.3348

Density
So Why R?

R enables us to

- work interactively
- explore and visualize data
- access, retrieve and/or generate data
- summarize and report into pdf, html, ...

making it the key language for statistical computing, and a preferred environment for many data analysts.
So Why R?

R has always been extensible via

- **C** via a bare-bones interface described in *Writing R Extensions*
- **Fortran** which is also used internally by R
- **Java** via *rJava* by Simon Urbanek
- **C++** but essentially at the bare-bones level of C

So while *in theory* this always worked – it was tedious *in practice*
Chambers (2008), opens Chapter 11 *Interfaces I: Using C and Fortran*:

*Since the core of R is in fact a program written in the C language, it’s not surprising that the most direct interface to non-R software is for code written in C, or directly callable from C. All the same, including additional C code is a serious step, with some added dangers and often a substantial amount of programming and debugging required. You should have a good reason.*
Chambers (2008), opens Chapter 11 *Interfaces I: Using C and Fortran*:

*Since the core of R is in fact a program written in the C language, it’s not surprising that the most direct interface to non-R software is for code written in C, or directly callable from C. All the same, including additional C code is a serious step, with some added dangers and often a substantial amount of programming and debugging required. You should have a good reason.*
Chambers proceeds with this rough map of the road ahead:

- **Against:**
  - It’s more work
  - Bugs will bite
  - Potential platform dependency
  - Less readable software

- **In Favor:**
  - New and trusted computations
  - Speed
  - Object references
Why Extend R?

The *Why?* boils down to:

- **speed** Often a good enough reason for us ... and a focus for us in this workshop.
- **new things** We can bind to libraries and tools that would otherwise be unavailable in R
- **references** Chambers quote from 2008 foreshadowed the work on the new *Reference Classes* now in R and built upon via Rcpp Modules, Rcpp Classes (and also RcppR6)
Overview: Why C++?
Why C++?

- Asking Google leads to about 52 million hits.
- Wikipedia: C++ is a statically typed, free-form, multi-paradigm, compiled, general-purpose, powerful programming language
- C++ is industrial-strength, vendor-independent, widely-used, and still evolving
- In science & research, one of the most frequently-used languages: If there is something you want to use / connect to, it probably has a C/C++ API
- As a widely used language it also has good tool support (debuggers, profilers, code analysis)
Scott Meyers: View C++ as a federation of languages

- C provides a rich inheritance and interoperability as Unix, Windows, ... are all build on C.
- Object-Oriented C++ (maybe just to provide endless discussions about exactly what OO is or should be)
- Templated C++ which is mighty powerful; template meta programming unequalled in other languages.
- The Standard Template Library (STL) is a specific template library which is powerful but has its own conventions.
- C++11 (and C++14 and beyond) add enough to be called a fifth language.

NB: Meyers original list of four languages appeared years before C++11.
Why C++?

- Mature yet current
- Strong performance focus:
  - *You don’t pay for what you don’t use*
  - *Leave no room for another language between the machine level and C++*
- Yet also powerfully abstract and high-level
- C++11 is a big deal giving us new language features
- While there are complexities, Rcpp users are mostly shielded
Overview: Vision
Algorithm Interface

ABC: general (FORTRAN) algorithm

XABC: FORTRAN subroutine to provide interface between ABC and/or language and/or utility programs

XABC (INSTR, OUTSTR)

Input INSTR →

<table>
<thead>
<tr>
<th>&quot;X&quot;</th>
<th>&quot;Y&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;X&quot;</td>
<td>&quot;Y&quot;</td>
</tr>
</tbody>
</table>

Argument Names or Blank

Pointers/Values
R offers us the best of both worlds:

- **Compiled** code with
  - Access to proven libraries and algorithms in C/C++/Fortran
  - Extremely high performance (in both serial and parallel modes)

- **Interpreted** code with
  - An accessible high-level language made for *Programming with Data*
  - An interactive workflow for data analysis
  - Support for rapid prototyping, research, and experimentation
Why Rcpp?

- *Easy to learn* as it really does not have to be that complicated – we will see numerous few examples
- *Easy to use* as it avoids build and OS system complexities thanks to the R infrastructure
- *Expressive* as it allows for *vectorised C++* using *Rcpp Sugar*
- *Seamless* access to all R objects: vector, matrix, list, S3/S4/RefClass, Environment, Function, …
- *Speed gains* for a variety of tasks Rcpp excels precisely where R struggles: loops, function calls, …
- *Extensions* greatly facilitates access to external libraries using eg *Rcpp modules*
Overview: Speed
Five different ways to compute $1/(1 + x)$:

```r
f <- function(n, x=1) for(i in 1:n) x <- 1/(1+x)
g <- function(n, x=1) for(i in 1:n) x <- (1/(1+x))
h <- function(n, x=1) for(i in 1:n) x <- (1+x)^(-1)
j <- function(n, x=1) for(i in 1:n) x <- {1/{1+x}}
k <- function(n, x=1) for(i in 1:n) x <- 1/{1+x}
library(rbenchmark)
N <- 1e5
benchmark(f(N,1),g(N,1),h(N,1),j(N,1),k(N,1),order="relative")
```
## Speed Example 1 (due to Christian Robert)

<table>
<thead>
<tr>
<th>test</th>
<th>replications</th>
<th>elapsed</th>
<th>relative</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 k(N, 1)</td>
<td>100</td>
<td>6.435</td>
<td>1.000</td>
</tr>
<tr>
<td>1 f(N, 1)</td>
<td>100</td>
<td>6.609</td>
<td>1.027</td>
</tr>
<tr>
<td>2 g(N, 1)</td>
<td>100</td>
<td>7.757</td>
<td>1.205</td>
</tr>
<tr>
<td>4 j(N, 1)</td>
<td>100</td>
<td>7.882</td>
<td>1.225</td>
</tr>
<tr>
<td>3 h(N, 1)</td>
<td>100</td>
<td>11.766</td>
<td>1.828</td>
</tr>
</tbody>
</table>
Adding a C++ variant is easy:

```cpp
cppFunction("
    double m(int n, double x) {
        for (int i=0; i<n; i++)
            x = 1 / (1+x);
        return x;
    }
")
```

(We will learn more about `cppFunction()` later).
## test replications elapsed relative

<table>
<thead>
<tr>
<th>#</th>
<th>Description</th>
<th>Replications</th>
<th>Elapsed Time</th>
<th>Relative Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>m(N, 1)</td>
<td>100</td>
<td>0.170</td>
<td>1.000</td>
</tr>
<tr>
<td>1</td>
<td>f(N, 1)</td>
<td>100</td>
<td>6.854</td>
<td>40.318</td>
</tr>
<tr>
<td>5</td>
<td>k(N, 1)</td>
<td>100</td>
<td>7.811</td>
<td>45.947</td>
</tr>
<tr>
<td>4</td>
<td>j(N, 1)</td>
<td>100</td>
<td>9.183</td>
<td>54.018</td>
</tr>
<tr>
<td>2</td>
<td>g(N, 1)</td>
<td>100</td>
<td>9.489</td>
<td>55.818</td>
</tr>
<tr>
<td>3</td>
<td>h(N, 1)</td>
<td>100</td>
<td>11.725</td>
<td>68.971</td>
</tr>
</tbody>
</table>
Consider a function defined as

\[
   f(n) \text{ such that } \begin{cases} 
   n & \text{when } n < 2 \\
   f(n - 1) + f(n - 2) & \text{when } n \geq 2
   \end{cases}
\]
R implementation and use:

f <- function(n) {
  if (n < 2) return(n)
  return(f(n-1) + f(n-2))
}

# Using it on first 11 arguments
sapply(0:10, f)

# [1] 0 1 1 2 3 5 8 13 21 34 55
Speed Example 2 (due to StackOverflow)

Timing:

```r
library(rbenchmark)
benchmark(f(10), f(15), f(20))[,1:4]
```

<table>
<thead>
<tr>
<th></th>
<th>test</th>
<th>replications</th>
<th>elapsed</th>
<th>relative</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>f(10)</td>
<td>100</td>
<td>0.030</td>
<td>1.000</td>
</tr>
<tr>
<td>2</td>
<td>f(15)</td>
<td>100</td>
<td>0.335</td>
<td>11.167</td>
</tr>
<tr>
<td>3</td>
<td>f(20)</td>
<td>100</td>
<td>3.517</td>
<td>117.233</td>
</tr>
</tbody>
</table>
int g(int n) {
    if (n < 2) return(n);
    return(g(n-1) + g(n-2));
}

deployed as

Rcpp::cppFunction("int g(int n) {
    if (n < 2) return(n);
    return(g(n-1) + g(n-2)); }")
sapply(0:10, g)

## [1] 0 1 1 2 3 5 8 13 21 34 55
Speed Example 2 (due to StackOverflow)

Timing:

```r
Rcpp::cppFunction("int g(int n) {
    if (n < 2) return(n);
    return(g(n-1) + g(n-2)); }")
library(rbenchmark)
benchmark(f(20), g(20), order="relative")[,1:4]
```

```
##          test replications elapsed relative
##     2 g(20)        100   0.011   1.000
##     1 f(20)        100   3.776 343.273
```

A nice gain of a few orders of magnitude.
Run-time performance is just one example.

*Time to code* is another metric.

We feel quite strongly that helps you code more succinctly, leading to fewer bugs and faster development.

A good environment helps. RStudio integrates R and C++ development quite nicely (eg the compiler error message parsing is very helpful) and also helps with package building.
What Next?
Programming with C++

- C++ Basics
- Debugging
- Best Practices

and then on to Rcpp itself
C++ Basics
Compiled not Interpreted

Need to compile and link

```c
#include <cstdio>

int main(void) {
    printf("Hello, world!\n");
    return 0;
}
```
Compiled not Interpreted

Or streams output rather than printf

```
#include <iostream>

int main(void) {
    std::cout << "Hello, world!" << std::endl;
    return 0;
}
```
g++  -o will compile and link

We will now look at an examples with explicit linking.
#include <cstdio>

#define MATHLIB_STANDALONE
#include <Rmath.h>

int main(void) {
    printf("N(0,1) 95th percentile %9.8f\n", qnorm(0.95, 0.0, 1.0, 1, 0));
}

We may need to supply:

- *header location* via `-I`,
- *library location* via `-L`,
- *library* via `-llibraryname`

```
g++ -I/usr/include -c qnorm_rmath.cpp
g++ -o qnorm_rmath qnorm_rmath.o -L/usr/lib -lRmath
```
Statically Typed

- R is dynamically typed: \( x \leftarrow 3.14; \ x \leftarrow "foo" \) is valid.
- In C++, each variable must be declared before first use.
- Common types are \texttt{int} and \texttt{long} (possibly with \texttt{unsigned}), \texttt{float} and \texttt{double}, \texttt{bool}, as well as \texttt{char}.
- No standard string type, though \texttt{std::string} is close.
- All these variables types are scalars which is fundamentally different from R where everything is a vector.
- \texttt{class} (and \texttt{struct}) allow creation of composite types; classes add behaviour to data to form objects.
- Variables need to be declared, cannot change
C++ Basics: A Better C
C++ is a Better C

- control structures similar to what R offers: for, while, if, switch
- functions are similar too but note the difference in positional-only matching, also same function name but different arguments allowed in C++
- pointers and memory management: very different, but lots of issues people had with C can be avoided via STL (which is something Rcpp promotes too)
- sometimes still useful to know what a pointer is ...
This is a second key feature of C++, and it does it differently from S3 and S4.

```cpp
struct Date {
    unsigned int year;
    unsigned int month;
    unsigned int day
};

struct Person {
    char firstname[20];
    char lastname[20];
    struct Date birthday;
    unsigned long id;
};
```
Object-orientation in the C++ sense matches data with code operating on it:

```cpp
class Date {
private:
    unsigned int year;
    unsigned int month;
    unsigned int date;
public:
    void setDate(int y, int m, int d);
    int getDay();
    int getMonth();
    int getYear();
}
```
Generic Programming and the STL

The STL promotes generic programming.

For example, the sequence container types vector, deque, and list all support

- `push_back()` to insert at the end;
- `pop_back()` to remove from the front;
- `begin()` returning an iterator to the first element;
- `end()` returning an iterator to just after the last element;
- `size()` for the number of elements;

but only list has `push_front()` and `pop_front()`.

Other useful containers: set, multiset, map and multimap.
Traversal of containers can be achieved via *iterators* which require suitable member functions *begin()* and *end()*:

```cpp
std::vector<double>::const_iterator si;
for (si=s.begin(); si != s.end(); si++)
    std::cout << *si << std::endl;
```
Another key STL part are *algorithms*:

```cpp
double sum = accumulate(s.begin(), s.end(), 0);
```

Some other STL algorithms are

- `find` finds the first element equal to the supplied value
- `count` counts the number of matching elements
- `transform` applies a supplied function to each element
- `for_each` sweeps over all elements, does not alter
- `inner_product` inner product of two vectors
Template programming provides a ‘language within C++’: code gets evaluated during compilation.

One of the simplest template examples is

```cpp
template <typename T>
const T& min(const T& x, const T& y) {
    return y < x ? y : x;
}
```

This can now be used to compute the minimum between two `int` variables, or `double`, or in fact any admissible type providing an operator\(<()\) for less-than comparison.
Another template example is a class squaring its argument:

```cpp
template <typename T>
class square : public std::unary_function<T,T> {
  public:
    T operator()(T t) const {
      return t*t;
    }
};
```

which can be used along with STL algorithms:

```cpp
transform(x.begin(), x.end(), square);
```
Further Reading

Books by Meyers are excellent

I also like the (free) C++ Annotations

C++ FAQ

Resources on StackOverflow such as

• general info and its links, eg
• booklist
Debugging
Some tips:

- Generally painful, old-school `printf()` still pervasive
- Debuggers go along with compilers: `gdb` for `gcc` and `g++`; `lldb` for the clang / llvm family
- Extra tools such as `valgrind` helpful for memory debugging
- “Sanitizer” (ASAN/UBSAN) in newer versions of `g++` and `clang++`
Best Practices
Version control: highly recommended to become familiar with git or svn

Editor: *in the long-run*, recommended to learn productivity tricks for one editor: emacs, vi, eclipse, RStudio, …