Higher-Performance R Programming with C++ Extensions

Part 1: Introduction

Dirk Eddelbuettel

June 28 and 29, 2017

University of Zürich & ETH Zürich

Overview

What Are We Doing Today and Tomorrow?

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*, ...

Before the Why/How/What

Maybe some mutual introductions?

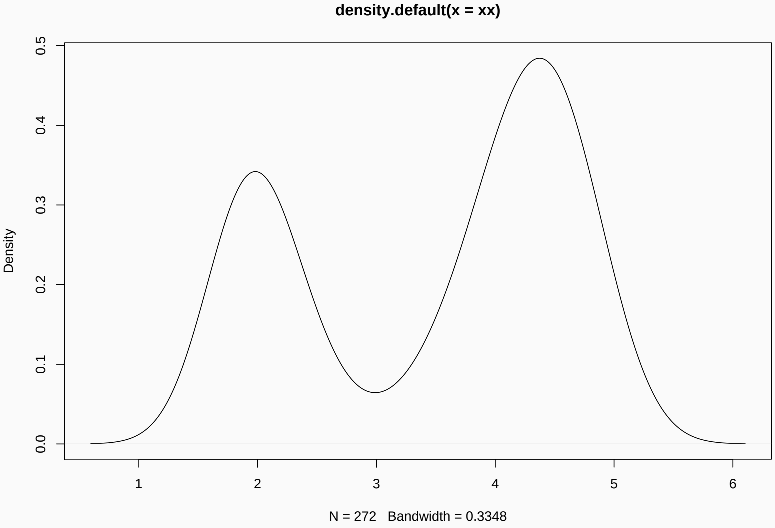
- Your background (academic, industry, ...)
- R experience (beginner, intermediate, advanced, ...)
- Created / modified any R packages ?
- C and/or C++ experience ?
- Main interest in Rcpp: speed, extensions, ..., ?
- Following `rcpp-devel` or `r-devel` ?

Why R?

A Simple Example

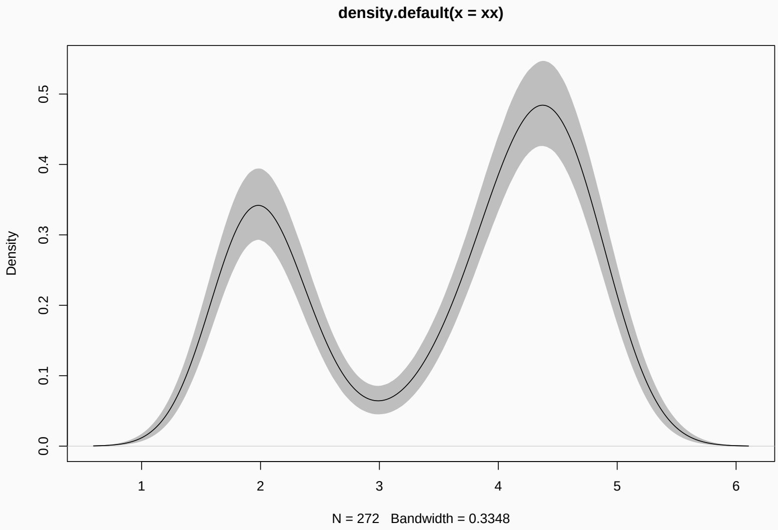
```r
xx <- faithful[,"eruptions"]
fit <- density(xx)
plot(fit)
```

A Simple Example



```
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



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.

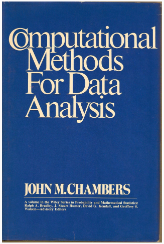
So Why R?

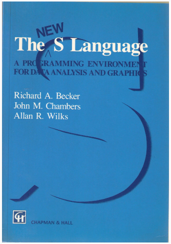
R has always been extensible via

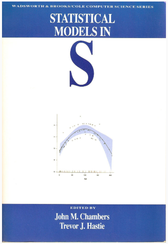
- C via a bare-bones interface described in *Writing R Extensions*
- Fortran which is also used internally by R
- Java via **`rJava`** by Simon Urbanek
- C++ but essentially at the bare-bones level of C

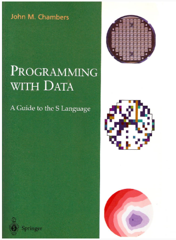
So while *in theory* this always worked – it was tedious *in practice*

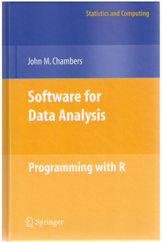
WHY R? : *PROGRAMMING WITH DATA* FROM 1977 TO 2016

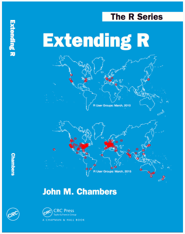












Thanks to John Chambers for high-resolution cover images. The publication years are, respectively, 1977, 1988, 1992, 1998, 2008 and 2016.

Why Extend R?

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.*

Why Extend R?

Chambers proceeds with this rough map of the road ahead:

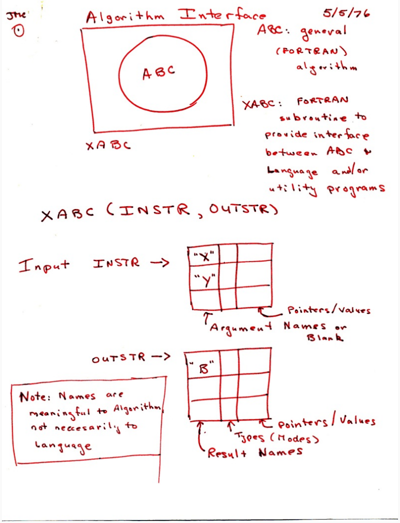
- Against:
    - It’s more work
    - Bugs will bite
    - Potential platform dependency
    - Less readable software
- In Favor:
    - New and trusted computations
    - Speed
    - Object references

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 *Reference Classes* now in R and built upon via Rcpp Modules, Rcpp Classes (and also RcppR6)

Bell Labs, May 1976



Interface Vision

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
  - A high-level language made for *Programming with Data*
  - An interactive workflow for data analysis
  - Support for rapid prototyping, research, and experimentation

And Why C++?

- Asking Google leads to tens of million of 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)

Why C++?

Scott Meyers: *View C++ as a federation of languages*

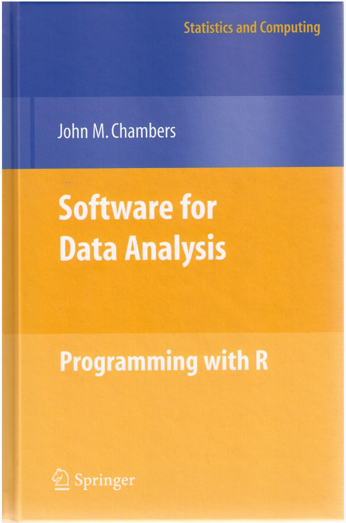
- *C* provides a rich inheritance and interoperability as Unix, Windows, … are all build on C.
- *Object-Oriented C++* (maybe just to provide endless discussions about exactly what OO is or should be)
- *Templated C++* which is mighty powerful; template meta programming unequalled in other languages.
- *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 + C++14 are a big deal giving us new language features
- While there are complexities, Rcpp users are mostly shielded

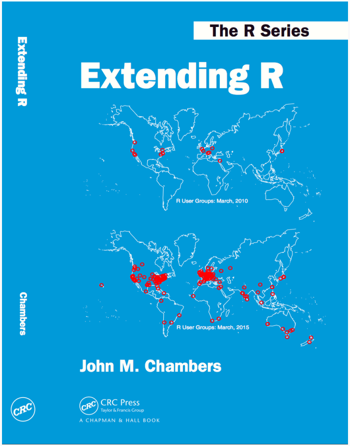
Interlude



Chambers (2008) Software For Data Analysis

Chapters 10 and 11 devoted to *Interfaces I: C and Fortran* and *Interfaces II: Other Systems*.

John Chambers



Chambers (2016) Extending R

An entire book about this with *concrete* Python, Julia and C++ code and examples

Chambers 2016, Chapter 1

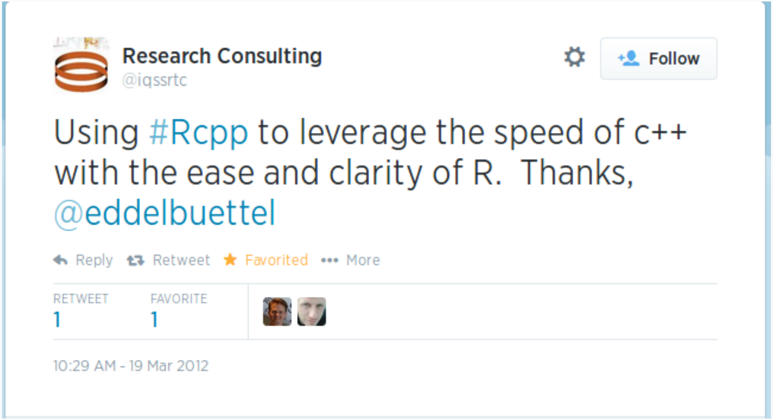
- *Everything that exists in R is an object*
- *Everything happens in R is a function call*
- *Interfaces to other software are part of R*

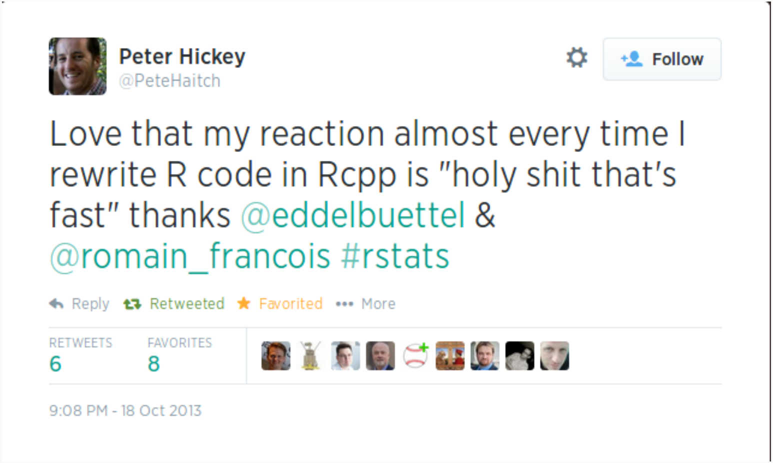
John Chambers

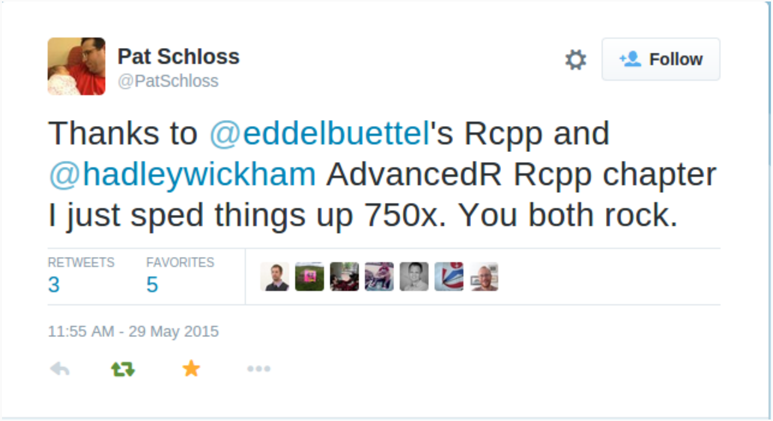
Chambers 2016, Chapter 4

> *The fundamental lesson about programming in the large is that requires a correspondingly broad and flexible response. In particular, no single language or software system os likely to be ideal for all aspects. Interfacing multiple systems is the essence. Part IV explores the design of of interfaces from R.*

Why Rcpp? Some Tweets


Research Consulting
@iqssrtc
Follow
Using #Rcpp to leverage the speed of c++ with the ease and clarity of R. Thanks, @eddelbuettel
Reply
Retweet
Favorited
More
RETWEET
1
FAVORITE
1
10:29 AM - 19 Mar 2012


Peter Hickey
@PeteHaitch
Follow
Love that my reaction almost every time I rewrite R code in Rcpp is "holy shit that's fast" thanks @eddelbuettel & @romain_francois #rstats
Reply
Retweeted
Favorited
More
RETWEETS
6
FAVORITES
8
9:08 PM - 18 Oct 2013


Pat Schloss
@PatSchloss
Follow
Thanks to @eddelbuettel's Rcpp and @hadleywickham AdvancedR Rcpp chapter I just sped things up 750x. You both rock.
RETWEETS
3
FAVORITES
5
11:55 AM - 29 May 2015


Rich FitzJohn
@rgfitzjohn

Follow

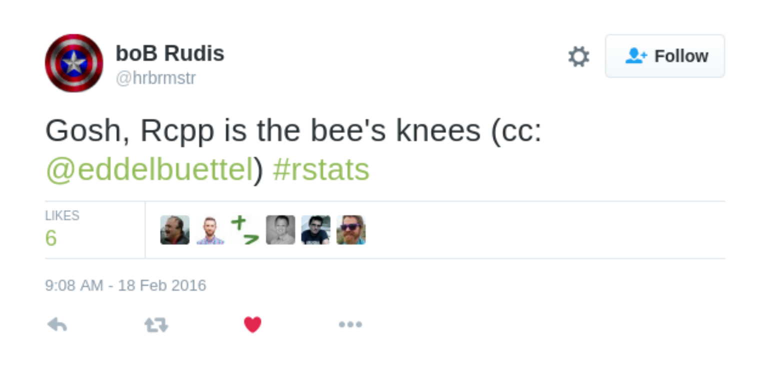
Writing some code using #rstats plain C API and realising/remembering quite how much work Rcpp saves - thanks @eddelbuettel

RETWEETS 5 FAVORITES 8

5:45 PM - 6 Mar 2015


Romain François
@romain_francois
Following
"Rcpp is one of the 3 things that changed how I write #rstats code". @hadleywickham at #EARL2014
RETWEETS 3
FAVORITES 7
3:19 AM - 16 Sep 2014

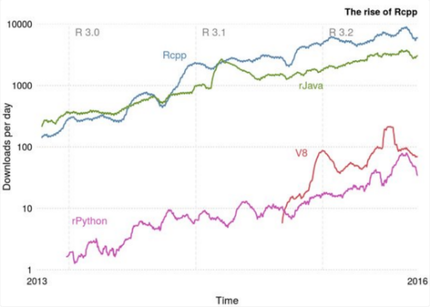

Karl Broman
@kwbroman
Following
@eddelbuettel @romain_francois Have I emphasized how much I ❤ #Rcpp?
LIKES
8
9:12 PM - 27 May 2016


boB Rudis
@hrbrmstr
Follow
Gosh, Rcpp is the bee's knees (cc: @eddelbuettel) #rstats
LIKES
6
9:08 AM - 18 Feb 2016

Colin Gillespie
@csgillespie

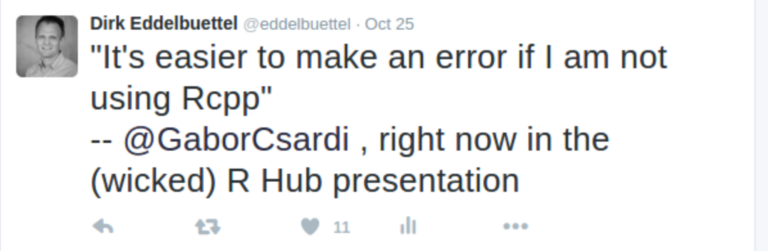
Following

The rise of Rcpp #rstats



RETWEETS 9 LIKES 15

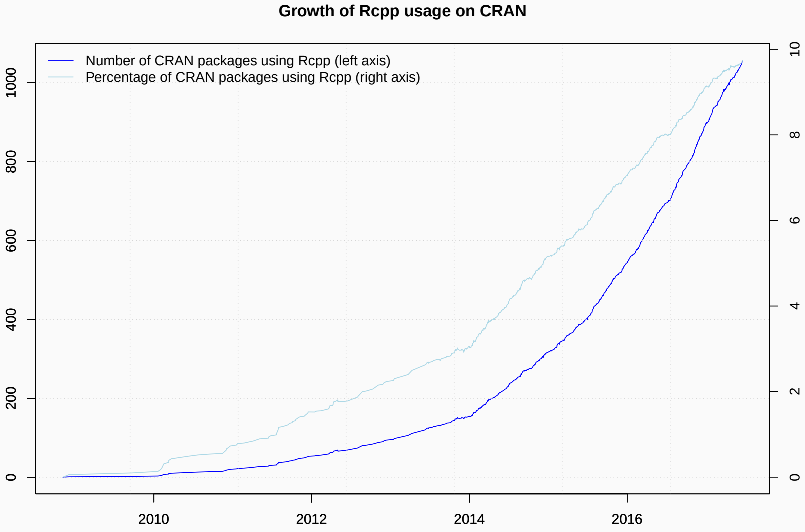
9:58 AM - 28 Apr 2016


Dirk Eddelbuettel @eddelbuettel · Oct 25
"It's easier to make an error if I am not using Rcpp"
-- @GaborCsardi , right now in the (wicked) R Hub presentation
11

Key points

- *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 infrastrucure
- *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*

Who Uses Rcpp?



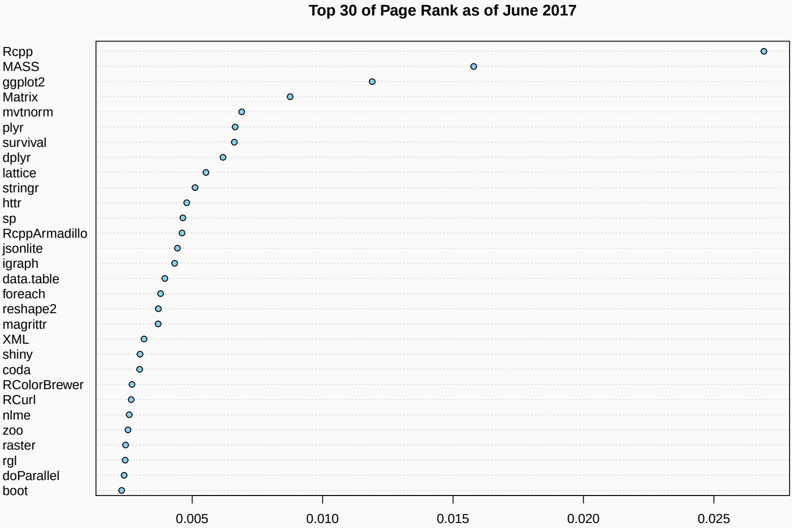
Data current as of June 17, 2017.

Pagerank

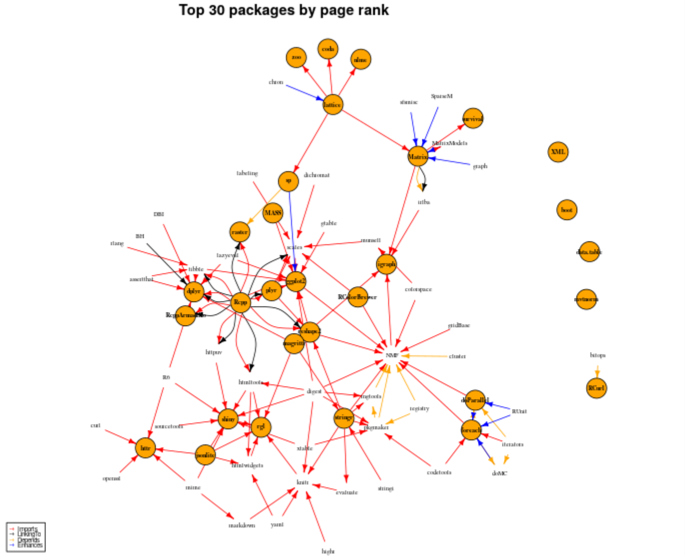
```r
library(pagerank)   # github.com/andrie/pagerank

cran <- "http://cloud.r-project.org"

pr <- compute_pagerank(cran)
round(100*pr[1:5], 3)
```

```
##    Rcpp    MASS ggplot2  Matrix mvtnorm
##   2.692   1.579   1.190   0.876   0.690
```

Pagerank



Top 30 of Page Rank as of June 2017
Rcpp
MASS
ggplot2
Matrix
mvtnorm
plyr
survival
dplyr
lattice
stringr
httr
sp
RcppArmadillo
jsonlite
igraph
data.table
foreach
reshape2
magrittr
XML
shiny
coda
RColorBrewer
RCurl
nlme
zoo
raster
rgl
doParallel
boot
0.005
0.010
0.015
0.020
0.025


Top 30 packages by page rank
zoo
coda
nlme
lattice
chron
survival
Matrix
graph
XML
boot
data.table
mvtnorm
RCurl
sp
MASS
raster
igraph
ggplot2
plyr
Rcpp
dplyr
RColorBrewer
reshape2
magrittr
NMF
shiny
rgl
httr
jsonlite
stringr
foreach
Imports
LinkingTo
Depends
Enhances

CRAN Proportion

```r
db <- tools::CRAN_package_db()   # R 3.4.0 or later
dim(db)
```

```
## [1] 10849    65
```

```r
## all Rcpp reverse depends
(c(n_rcpp <- length(tools::dependsOnPkgs("Rcpp", recursive=FALSE,
                                          installed=db)),
   n_compiled <- table(db[, "NeedsCompilation"])[["yes"]]))
```

```
## [1] 1057 2900
```

```r
## Rcpp percentage of packages with compiled code
n_rcpp / n_compiled
```

```
## [1] 0.364483
```

Speed

Speed Example 1 (due to Christian Robert)

Five different ways to compute $1/(1+x)$:

```
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)

```
##        test replications elapsed relative
## 1 f(N, 1)          100   0.612    1.000
## 5 k(N, 1)          100   0.612    1.000
## 2 g(N, 1)          100   0.613    1.002
## 4 j(N, 1)          100   0.615    1.005
## 3 h(N, 1)          100   0.798    1.304
```

Speed Example 1 (due to Christian Robert)

Adding a C++ variant is easy:

```
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).

Speed Example 1 (due to Christian Robert)

```
##        test replications elapsed relative
## 6 m(N, 1)          100   0.100     1.00
## 4 j(N, 1)          100   0.606     6.06
## 1 f(N, 1)          100   0.608     6.08
## 2 g(N, 1)          100   0.608     6.08
## 5 k(N, 1)          100   0.608     6.08
## 3 h(N, 1)          100   0.791     7.91
```

Speed Example 2 (due to StackOverflow)

Consider a function defined as

$$\mathrm{f}(n) \quad \text{such that} \quad \begin{cases} n & \text{when} \;\; n < 2 \\ \mathrm{f}(n-1) + \mathrm{f}(n-2) & \text{when} \;\; n \geq 2 \end{cases}$$

Speed Example 2 (due to StackOverflow)

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]
```

```
##     test replications elapsed relative
## 1 f(10)          100   0.009    1.000
## 2 f(15)          100   0.104   11.556
## 3 f(20)          100   1.125  125.000
```

Speed Example 2 (due to StackOverflow)

```
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   0.006    1.000
## 1 f(20)          100   1.178  196.333
```

A nice gain of a few orders of magnitude.

Another Angle on Speed

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 ?

- C++ Basics
- Debugging
- Best Practices

and then on to Rcpp itself

Need to compile and link

```
#include <cstdio>

int main(void) {
    printf("Hello, world!\n");
    return 0;
}
```

Or streams output rather than `printf`

```
#include <iostream>

int main(void) {
    std::cout << "Hello, world!" << std::endl;
    return 0;
}
```

Compiled not Interpreted

`g++ -o` will compile and link

We will now look at an examples with explicit linking.

Compiled not Interpreted

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

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.

```
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:

```
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();
}
```

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()`:

```
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*:

```
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

```
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:

```
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:

```
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: `git` or `svn` highly recommended

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