



EXTENDING R WITH C++

MOTIVATION, EXAMPLES, AND CONTEXT

Dirk Eddelbuettel

11 October 2017

Invited Presentation

Civis Analytics

Chicago, IL

OUTLINE

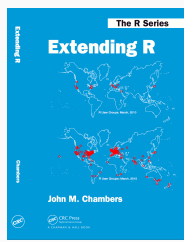
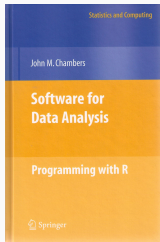
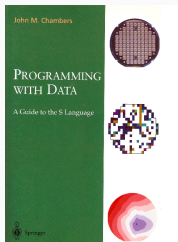
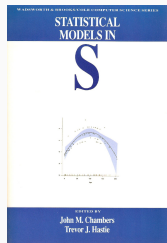
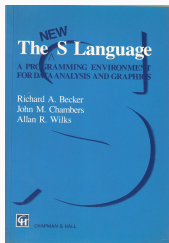
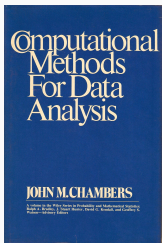
- Quick R Basics Reminder
- C++ in (way less than) a nutshell
- Extending R with C++ via Rpp
- Multi-Lingual Computing

Brief Bio

- PhD, MA Econometrics; MSc Ind.Eng. (Comp.Sci./OR)
- Finance Quant for 20+ years
- Open Source for 23+ years
 - Debian developer
 - R package author / contributor
- R and Statistics
 - JSS Associate Editor
 - R Foundation Board member
 - R Consortium ISC member

WHY R?

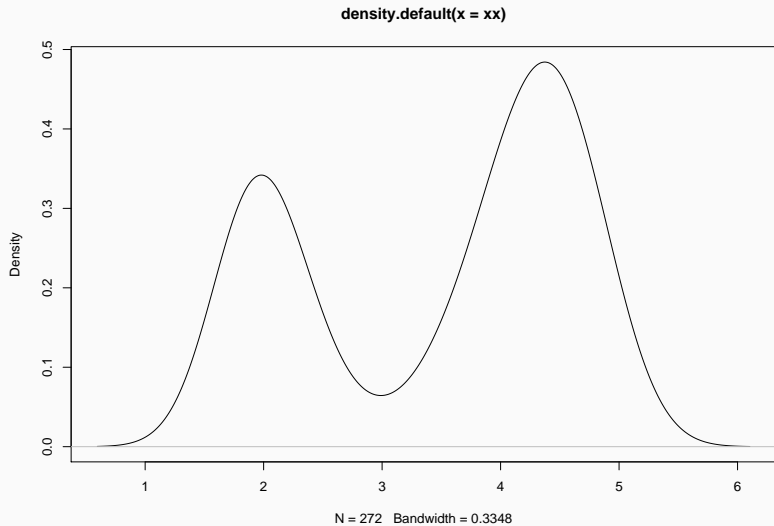
PROGRAMMING WITH DATA FROM 1977 TO 2016



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

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

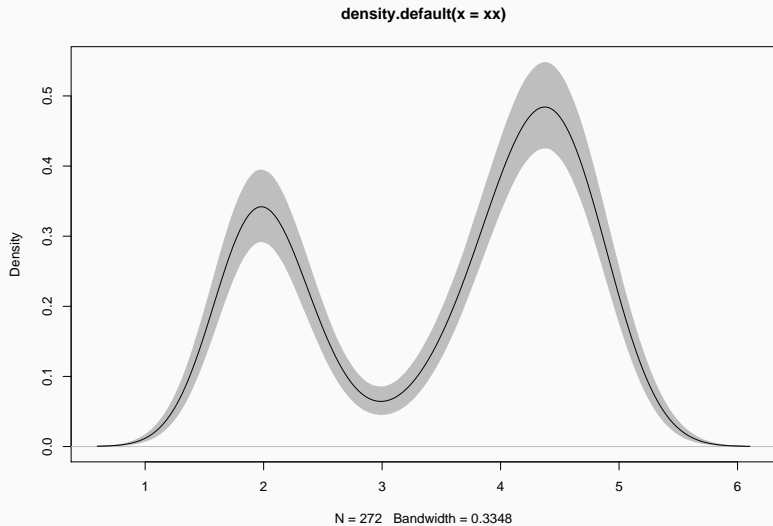
A SIMPLE EXAMPLE



A SIMPLE EXAMPLE - REFINED

```
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 is a fantastic *data* interface:

- powerful REPL, now also powerful IDE
- interfaces to import *from* anything:
 - databases
 - text or binary formats
- output to *report* in anything
 - first sweave, now (R)markdown and knitr
 - via pandoc export to any format
- Shiny dashboard are fabulous
- plays with others such as Jupiter notebooks too

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

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, C++17, ... a big deal giving us new language features
- While there are complexities, Rcpp users are mostly shielded

C++ IN TOO LITTLE TIME

Need to compile and link

```
#include <stdio>

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;  
}
```

`g++ -o` will compile and link

We will now look at an examples with explicit linking.


```
#include <stdio>

#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
```

- 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

- 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's different 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 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();  
}
```

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()`:

```
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);
```

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](#)

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++`

EXTENDING R WITH C++

Three key functions

- `evalCpp()`
- `sourceCpp()`
- `cppFunction()`

BASIC USAGE: EVALCPP()

`evalCpp()` evaluates a single C++ expression. Includes and dependencies can be declared.

This allows us to quickly check C++ constructs.

```
library(Rcpp)
```

```
evalCpp("2 + 2")      # simple test
```

```
## [1] 4
```

```
evalCpp("std::numeric_limits<double>::max()")
```

```
## [1] 1.797693e+308
```


BASIC USAGE: CPPFUNCTION()

`cppFunction()` creates, compiles and links a C++ file, and creates an R function to access it.

```
cppFunction("
  int simpleExample() {
    int x = 10;
    return x;
 }")
simpleExample() # same identifier as C++ function
```

BASIC USAGE: CPPFUNCTION()

`cppFunction()` creates, compiles and links a C++ file, and creates an R function to access it.

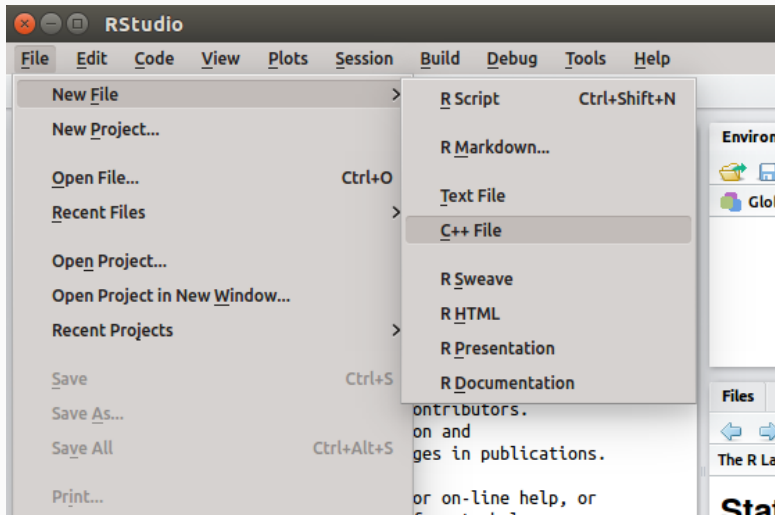
```
cppFunction("
    int exampleCpp11() {
        auto x = 10;
        return x;
    }", plugins=c("cpp11"))
exampleCpp11() # same identifier as C++ function
```

`sourceCpp()` is the actual workhorse behind `evalCpp()` and `cppFunction()`. It is described in more detail in the [package vignette Rcpp-attributes](#).

`sourceCpp()` builds on and extends `cxxfunction()` from package `inline`, but provides even more ease-of-use, control and helpers – freeing us from boilerplate scaffolding.

A key feature are the plugins and dependency options: other packages can provide a plugin to supply require compile-time parameters (cf `RcppArmadillo`, `RcppEigen`, `RcppGSL`).

BASIC USAGE: RSTUDIO



BASIC USAGE: RSTUDIO (CONT'ED)

The following file gets created:

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

// This is a simple example of exporting a C++ function to R. You can
// source this function into an R session using the Rcpp::sourceCpp
// function (or via the Source button on the editor toolbar). ...

// [[Rcpp::export]]
NumericVector timesTwo(NumericVector x) { return x * 2; }

// You can include R code blocks in C++ files processed with sourceCpp
// (useful for testing and development). The R code will be automatically
// run after the compilation.

/** R
timesTwo(42)
*/
```

So what just happened?

- We defined a simple C++ function
- It operates on a numeric vector argument
- We asked Rcpp to 'source it' for us
- Behind the scenes Rcpp creates a wrapper
- Rcpp then compiles, links, and loads the wrapper
- The function is available in R under its C++ name

Package are *the* standard unit of R code organization.

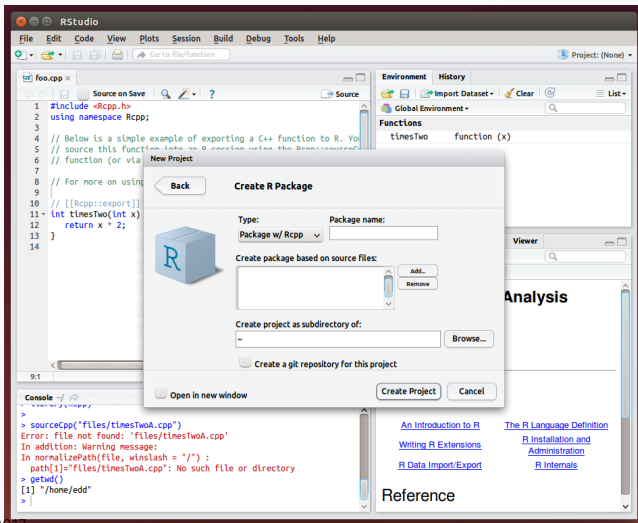
Creating packages with Rcpp is easy; an empty one to work from can be created by `Rcpp.package.skeleton()`

The vignette [Rcpp-packages](#) has fuller details.

As of October 11, 2017, there are 1185 packages on CRAN which use Rcpp, and a further 91 on BioConductor — with working, tested, and reviewed examples.

PACKAGES AND RCPP

Best way to organize R code with Rcpp is via a package:



The screenshot shows the RStudio interface with a C++ file named `foo.cpp` open. The code includes Rcpp headers and defines a function `tinesTwo` that takes an integer `x` and returns `x * 2`. A dialog box titled "Create R Package" is overlaid on the editor. The dialog has a "Back" button and a "Create R Package" title. It contains the following fields and options:

- Type:** A dropdown menu set to "Package w/ Rcpp".
- Package name:** An empty text input field.
- Create package based on source files:** An empty list box with "Add..." and "Remove" buttons.
- Create project as subdirectory of:** A text input field containing a tilde (~) and a "Browse..." button.
- Create a git repository for this project**
- Open in new window**
- Create Project** and **Cancel** buttons.

The background shows the R console with the following output:

```
> sourceCpp("files/tinesTwoA.cpp")
Error: file not found: 'files/tinesTwoA.cpp'
In addition: Warning message:
In normalizePath(file, winslash = "/") :
  path[1]='files/tinesTwoA.cpp': No such file or directory
> getwd()
[1] "/home/edd"
```

On the right side of the RStudio interface, there is a "Viewer" pane with a search bar and a "Reference" section containing several links:

- [An Introduction to R](#)
- [Writing R Extensions](#)
- [R Data Import/Export](#)
- [The R Language Definition](#)
- [R Installation and Administration](#)
- [R Internals](#)

Rcpp.package.skeleton() and its derivatives. e.g.

RcppArmadillo.package.skeleton() create working packages.

```
// another simple example: outer product of a vector,  
// returning a matrix  
//  
// [[Rcpp::export]]  
arma::mat rcpparma_outerproduct(const arma::colvec & x) {  
    arma::mat m = x * x.t();  
    return m;  
}  
  
// and the inner product returns a scalar  
//  
// [[Rcpp::export]]  
double rcpparma_innerproduct(const arma::colvec & x) {  
    double v = arma::as_scalar(x.t() * x);  
    return v;  
}
```

NICE, BUT DOES IT *REALLY* WORK?

Something self-contained

- Let's talk random numbers!
- We'll look at a quick generator
- And wrap it in plain C / C++

RANDOM NUMBER

<

< PREV

RANDOM

NEXT >

>

```
int getRandomNumber()  
{  
    return 4; // chosen by fair dice roll.  
              // guaranteed to be random.  
}
```

<

< PREV

RANDOM

NEXT >

>

PERMANENT LINK TO THIS COMIC: [HTTPS://XKCD.COM/221/](https://xkcd.com/221/)

```
// cf https://xkcd.com/221/
//
//   "RFC 1149.5 specifies 4 as the "
//   "standard IEEE-vetted random number."

int getRandomNumber()
{
    return 4; // chosen by fair dice roll
             // guaranteed to be random
}
```

GETXKCDRNGDRAW()

```
#include <Rcpp.h>
#include <xkcdRng.h>

// [[Rcpp::export]]
int getXkcdRngDraw() {
    return getRandomNumber();
}
```

The screenshot displays the RStudio environment for developing the 'samplexkcdrng' package. The source editor shows the following C++ code:

```

1 #include <Rcpp.h>
2 #include <xkcdRng.h>
3
4 // [[Rcpp::export]]
5 int getXkcdRngDraw() {
6     return getRandomNumber();
7 }
8
9

```

The console shows the directory structure:

```

edd@brad:~/git/samplexkcdrng$ tree
.
├── DESCRIPTION
├── man
│   ├── samplexkcdrng-package.Rd
├── NAMESPACE
├── R
│   ├── RcppExports.R
├── samplexkcdrng.Rproj
├── src
│   ├── getXkcdRngDraw.cpp
│   ├── Makevars
│   ├── RcppExports.cpp
│   └── xkcdRng.h
└── 3 directories, 9 files
edd@brad:~/git/samplexkcdrng$

```

The terminal shows the installation process:

```

edd@brad:~/git/samplexkcdrng$ R CMD SHLIB2 ~/git/samplexkcdrng
** R
** preparing package for lazy loading
** help
*** installing help indices
** building package indices
** testing if installed package can be loaded
* DONE (samplexkcdrng)

```

The Environment pane shows the package is installed and ready to use:

```

=> roxygen2::roxygenize('.', roclats=c('rd'))
First time using roxygen2. Upgrading automatically...
Documentation completed
=> R CMD INSTALL --no-multiarch --with-keep.source samplexkcdrng

```

The Files pane shows the package files:

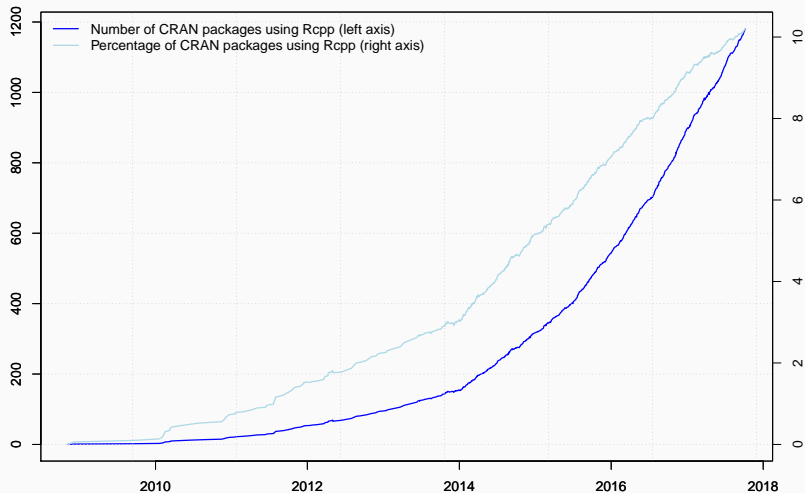
Name	Size	Modified
..		
.Rbuildignore	28 B	Oct 7, 2017, 9:57 PM
.Rhistory	0 B	Oct 7, 2017, 10:02 PM
DESCRIPTION	274 B	Oct 7, 2017, 10:05 PM
man		

WHAT DID WE DO?

- An unmodified piece of C / C++ code
- A simple interface function
- Rcpp does the rest

EMPIRICS OF USAGE

Growth of Rcpp usage on CRAN



Data current as of October 8, 2017.

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

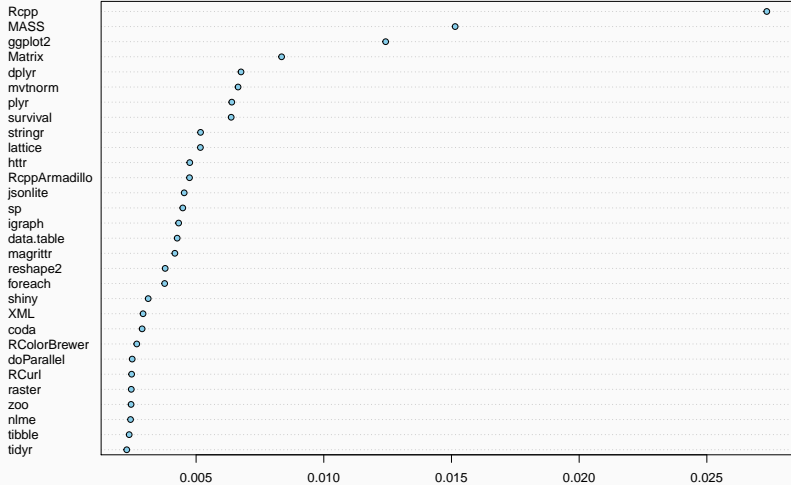
```
cran <- "http://cloud.r-project.org"
```

```
pr <- compute_pagerank(cran)
```

```
round(100*pr[1:5], 3)
```

##	Rcpp	MASS	ggplot2	Matrix	dplyr
##	2.735	1.514	1.242	0.835	0.676

Top 30 of Page Rank as of October 2017



CRAN PROPORTION

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

## [1] 11573

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

## [1] 1181 3050

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

## [1] 0.3872131
```

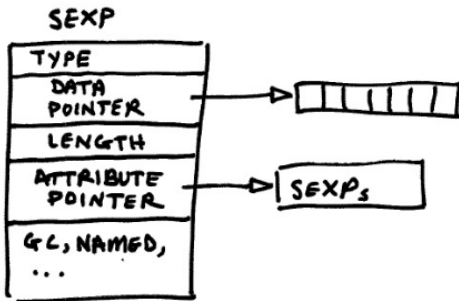
HOW DOES IT WORK?

THERE IS ONLY ONE INTERFACE FROM R

```
SEXP .Call("someFunction", SEXP a, SEXP b, SEXP c, ...)
```

SO WHAT IS A SEXP? IT'S COMPLICATED...

What's a SEXP?

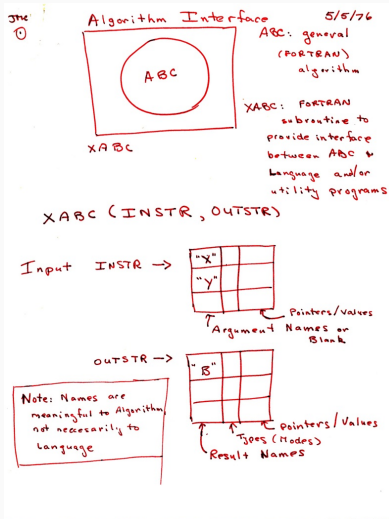


◀ ▶ ⏪ ⏩ 🔍

Source: Seth Falcon, <https://www.slideshare.net/userprimary/native-interfaces>, 2010.

- Rcpp mapping for every existing SEXP
- Meaning we can pass any R object
- And receive and modify it on the C++ side
- And return it, or newly created objects
- “Seamlessly”

INTERFACE VISION @ BELL LABS, MAY 1976



Source: John Chamber, personal communication

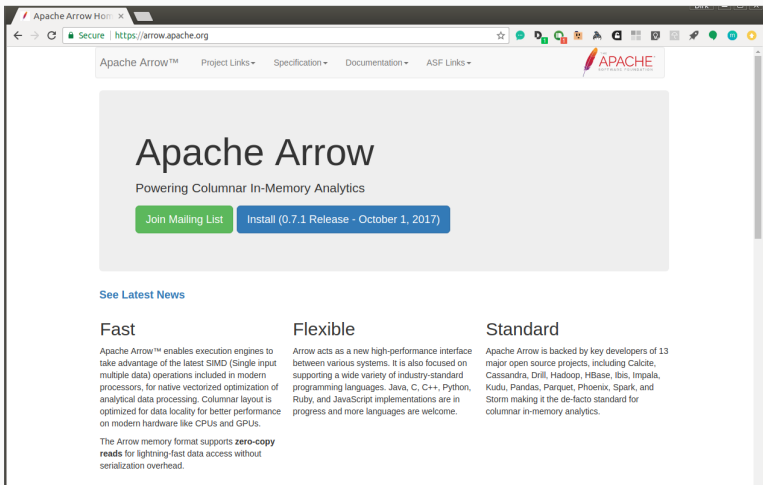
MULTI-LINGUAL COMPUTING

(SIMPLE) REAL-WORLD EXAMPLES

- The RcppCNPY packages uses a C library from GitHub to read and write NumPy files to/from R
- The RcppAnnoy package wraps Annoy by Erik Bernhardsson / Spotify: a clean and simple ANN library in C++ with an existing Python interface

- The reticulate package by JJ Allaire et al wraps Python
- It is already being used to access Tensorflow and Keras from R
- This may be general enough most (if not all) Python projects!

POTENTIALLY VERY EXCITING I



The screenshot shows the Apache Arrow homepage in a web browser. The browser's address bar displays "https://arrow.apache.org". The page features a navigation menu with links for "Project Links", "Specification", "Documentation", and "ASF Links". The Apache logo is visible in the top right corner. The main content area has a large heading "Apache Arrow" followed by the subtitle "Powering Columnar In-Memory Analytics". Below this are two buttons: "Join Mailing List" and "Install (0.7.1 Release - October 1, 2017)". A section titled "See Latest News" is followed by three columns: "Fast", "Flexible", and "Standard", each with a brief description of the technology's capabilities and performance benefits.

Apache Arrow™ Project Links Specification Documentation ASF Links

Apache Arrow

Powering Columnar In-Memory Analytics

[Join Mailing List](#) [Install \(0.7.1 Release - October 1, 2017\)](#)

[See Latest News](#)

Fast

Apache Arrow™ enables execution engines to take advantage of the latest SIMD (Single input multiple data) operations included in modern processors, for native vectorized optimization of analytical data processing. Columnar layout is optimized for data locality for better performance on modern hardware like CPUs and GPUs.

The Arrow memory format supports **zero-copy reads** for lightning-fast data access without serialization overhead.

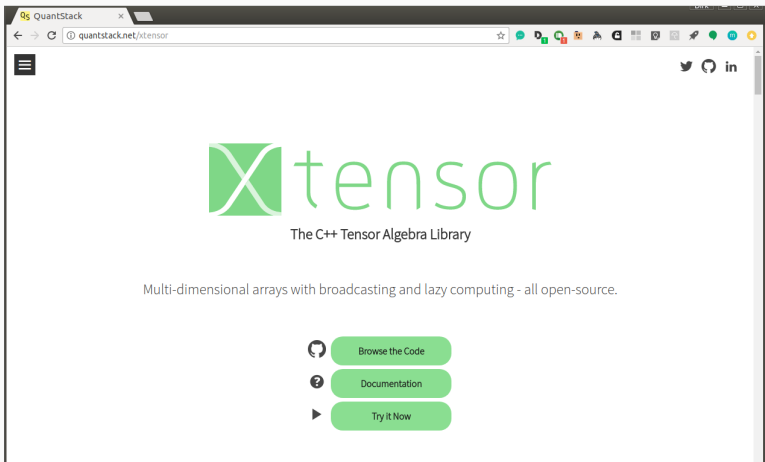
Flexible

Arrow acts as a new high-performance interface between various systems. It is also focused on supporting a wide variety of industry-standard programming languages. Java, C, C++, Python, Ruby, and JavaScript implementations are in progress and more languages are welcome.

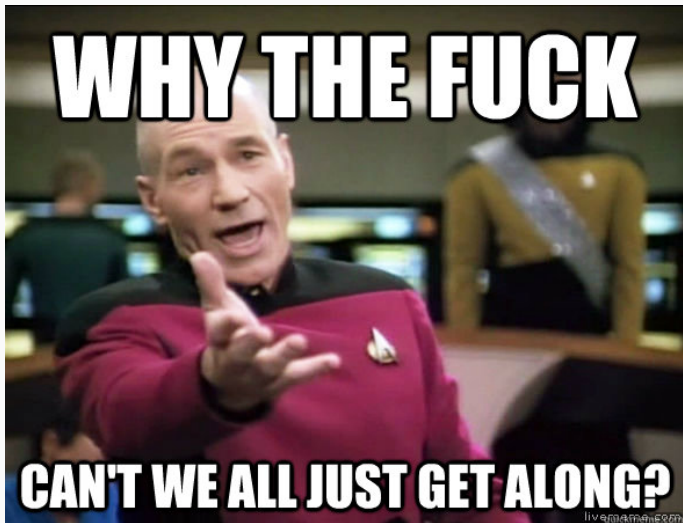
Standard

Apache Arrow is backed by key developers of 13 major open source projects, including Calcite, Cassandra, Drill, Hadoop, HBase, Ibis, Impala, Kudu, Pandas, Parquet, Phoenix, Spark, and Storm making it the de-facto standard for columnar in-memory analytics.

POTENTIALLY VERY EXCITING II



FINALLY



- We are wasting precious time fighting each other
- Particularly “license posts” are mostly toxic
- There are moral overtones I really dislike

Lars Wirzenius “Which license is the most free?”

Free software licences can be roughly grouped into permissive and copyleft ones. [...] A permissive licence lets you do things that a copyleft one forbids, so **clearly the permissive licence is more free**. A copyleft licence means software using it won't ever become non-free against the wills of the copyright holders, so **clearly a copyleft licence is more free than a permissive one**.

Both sides are both right and wrong, of course, which is why this argument will continue forever. [...]

If a discussion about the relative freedom of licence types becomes heated, step away. It's not worth participating anymore.

<http://yakking.branchable.com/posts/comparative-freeness/>

Reciprocal vs Non-reciprocal

I'm not keen on much of the language they were using to describe licenses, like "permissive" and "viral" to describe the Apache License and the GPL respectively. The GPL is a fine open source license that grants all the permissions needed by developer communities that adopt it. There is no sense in which it is not "permissive", so to use that term as an antonym to "copyleft" verges on abuse. I prefer to consider the degree to which open source licenses anticipate *reciprocal behaviour*.

The Apache License grants all its permissions without any expectation that those passing on the software to others will also pass on the same freedoms they enjoy. Copyleft licenses anticipate that developers *will* share the freedoms they enjoy as well as sharing the source code. So I term licenses like the Apache License "non-reciprocal" and those like the GPL "reciprocal".

There are many licenses in each of those two categories, with a wide variety of other attributes as well. For example, the BSD and MIT licenses are also non-reciprocal while the Eclipse License and Mozilla Public License are also reciprocal. There are also attributes other than reciprocity by which licenses might be classified, such as the nature of required attribution.

Source: <https://meshedinsights.com/2017/05/03/is-the-gpl-really-declining/>

SO LET'S DO MORE AWESOME

- Multi-lingual computing is real
- It's likely here to stay
- Let's not fight each other
- Rather:
 - Let's build more **awesome** things
 - That **interact** and **combine** better

Thank You!

<http://dirk.eddelbuettel.com/>

dirk@eddelbuettel.com

[@eddelbuettel](#)