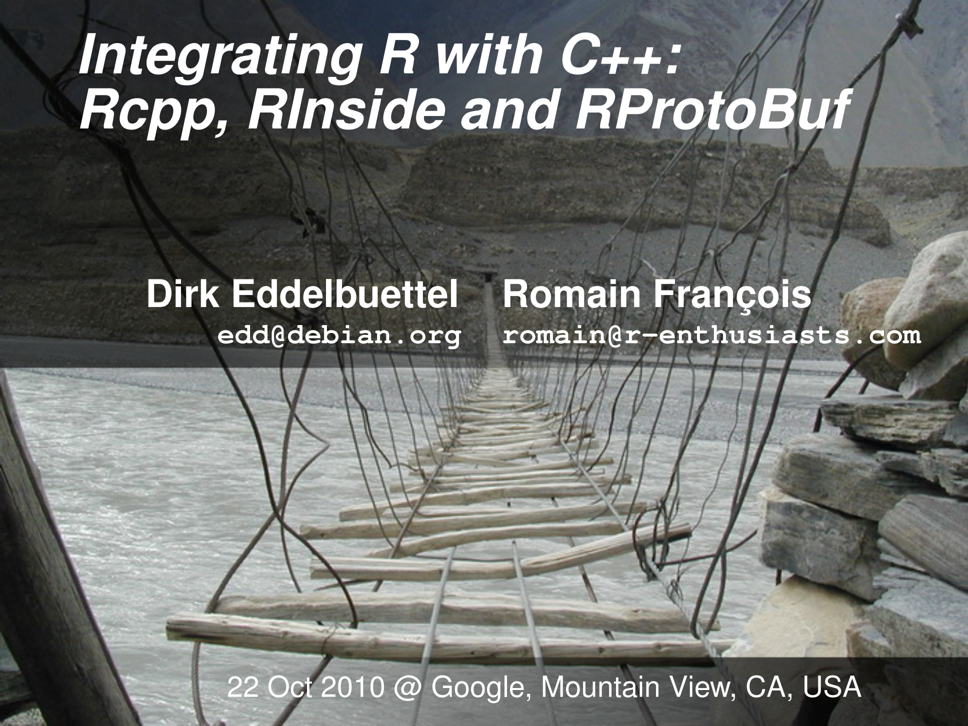


Integrating R with C++: Rcpp, RInside and RProtoBuf

Dirk Eddelbuettel
`edd@debian.org`

Romain François
`romain@r-enthusiasts.com`

22 Oct 2010 @ Google, Mountain View, CA, USA

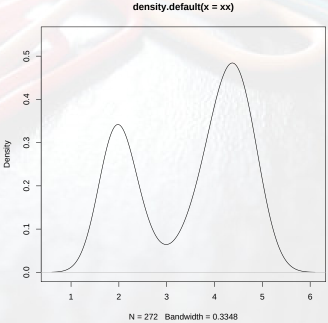
Preliminaries

- We assume a recent version of R such that `install.packages(c("Rcpp","RInside","inline"))` gets us current versions of the packages.
- RProtoBuf need the Protocol Buffer library and headers (which is not currently available on Windows / MinGW).
- All examples shown should work ’as is’ on Unix-alike OSs; most will also work on Windows *provided a complete R development environment*
- The Reference Classes examples assume R 2.12.0 and Rcpp 0.8.7.
- We may imply a `using namespace Rcpp;` in some of the C++ examples.

A Simple Example

Courtesy of Greg Snow via r-help

```
> xx <- faithful$eruptions
> fit <- density(xx)
> plot(fit)
```

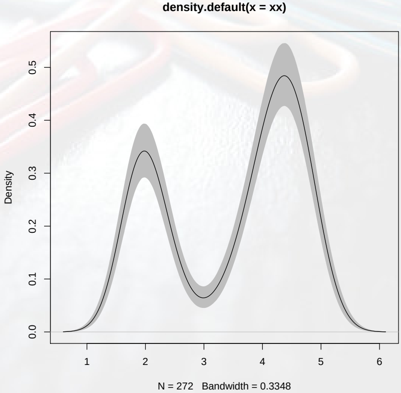


Standard R use: load some data, estimate a density, plot it.

A Simple Example

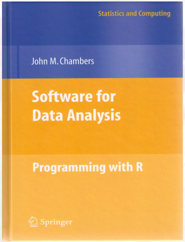
Now complete

```
> 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)
```



What other language can do that in seven statements?

Motivation



Chambers. *Software for Data Analysis: Programming with R*. Springer, 2008

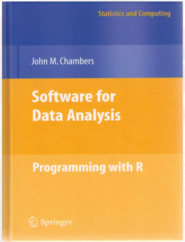
Chambers (2008) opens chapter 11 (*Interfaces I: Using C and Fortran)* with these words:

> *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.*

Motivation

Chambers (2008) opens chapter 11 (*Interfaces I: Using C and Fortran)* with these words:

> *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. *Software for Data Analysis: Programming with R*. Springer, 2008

Motivation

Chambers (2008) then 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

So is the deck stacked against us?

Le viaduc de Millau

Rcpp in a Nutshell

- The goal: *Seamless R and C++ Integration*
- R offers the `.Call()` interface operating on R internal `SEXP`
- We provide a natural object mapping between R and C++ using a class framework
- We enable immediate prototyping using extensions added to **inline**
- We also facilitate easy package building
- Extensions offer *e.g.* efficient templated linear algebra

Fine for Indiana Jones

R support for C/C++

- R is a C program
- R supports C++ out of the box, just use a `.cpp` file extension
- R exposes a API based on low level C functions and MACROS.
- R provides several calling conventions to invoke compiled code.

```
SEXP foo( SEXP x1, SEXP x2 ){
   ...
}
```

```
> .Call( "foo", 1:10, rnorm(10) )
```

.Call example

```
#include <R.h>
#include <Rdefines.h>
extern "C" SEXP vectorfoo(SEXP a, SEXP b){
  int i, n;
  double *xa, *xb, *xab; SEXP ab;
  PROTECT(a = AS_NUMERIC(a));
  PROTECT(b = AS_NUMERIC(b));
  n = LENGTH(a);
  PROTECT(ab = NEW_NUMERIC(n));
  xa=NUMERIC_POINTER(a); xb=NUMERIC_POINTER(b);
  xab = NUMERIC_POINTER(ab);
  double x = 0.0, y = 0.0 ;
  for (i=0; i<n; i++) xab[i] = 0.0;
  for (i=0; i<n; i++) {
    x = xa[i]; y = xb[i];
    res[i] = (x < y) ? x*x : -(y*y);
  }
  UNPROTECT(3);
  return(ab);
}
```

.Call example: character vectors

```
> c( "foo", "bar" )
```

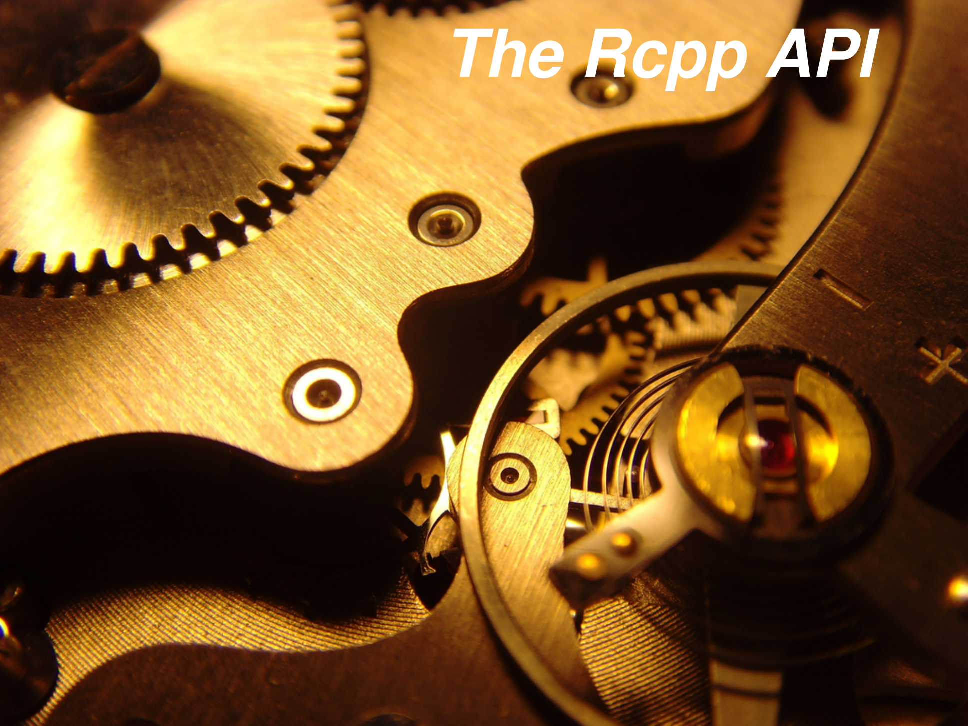
```
#include <R.h>
#include <Rdefines.h>
extern "C" SEXP foobar(){
  SEXP res = PROTECT(allocVector(STRSXP, 2));
  SET_STRING_ELT( res, 0, mkChar( "foo" ) ) ;
  SET_STRING_ELT( res, 1, mkChar( "bar" ) ) ;
  UNPROTECT(1) ;
  return res ;
}
```

.Call example: calling an R function

```
> eval( call( "rnorm", 3L, 10.0, 20.0 ) )
```

```
#include <R.h>
#include <Rdefines.h>
extern "C" SEXP callback(){
  SEXP call = PROTECT( LCONS( install("rnorm"),
    CONS( ScalarInteger( 3 ),
      CONS( ScalarReal( 10.0 ),
       CONS( ScalarReal( 20.0 ), R_NilValue )
     )
   )
  ) );
  SEXP res = PROTECT(eval(call, R_GlobalEnv)) ;
  UNPROTECT(2) ;
  return res ;
}
```

The Rcpp API



The Rcpp API

- Encapsulation of R objects (`SEXP`) into C++ classes: `NumericVector`, `IntegerVector`, ..., `Function`, `Environment`, `Language`, ...
- Conversion from R to C++ : `as`
- Conversion from C++ to R : `wrap`
- Interoperability with the Standard Template Library (STL)

The Rcpp API : classes

Rcpp class	R `typeof`
`Integer(Vector|Matrix)`	`integer` vectors and matrices
`Numeric(Vector|Matrix)`	`numeric` ...
`Logical(Vector|Matrix)`	`logical` ...
`Character(Vector|Matrix)`	`character` ...
`Raw(Vector|Matrix)`	`raw` ...
`Complex(Vector|Matrix)`	`complex` ...
`List`	`list` (aka generic vectors) ...
`Expression(Vector|Matrix)`	`expression` ...
`Environment`	`environment`
`Function`	`function`
`XPtr`	`externalptr`
`Language`	`language`
`S4`	`S4`
...	...

The Rcpp API : numeric vectors

Create a vector:

```
SEXP x ;
NumericVector y( x ) ; // from a SEXP

// cloning (deep copy)
NumericVector z = clone<NumericVector>( y ) ;

// of a given size (all elements set to 0.0)
NumericVector y( 10 ) ;

// ... specifying the value
NumericVector y( 10, 2.0 ) ;

// ... with elements generated
NumericVector y( 10, ::Rf_unif_rand ) ;

// with given elements
NumericVector y = NumericVector::create( 1.0, 2.0 ) ;
```

The Rcpp API : environments

```
Environment::global_env() ;
Environment::empty_env() ;
Environment::base_env() ;
Environment::base_namespace() ;
Environment::Rcpp_namespace() ;

Environment env( 2 ) ;

Environment env( "package:Rcpp" ) ;

Environment Rcpp = Environment::Rcpp_namespace() ;
Environment env = Rcpp.parent() ;
Environment env = Rcpp.new_child(true) ;

Environment Rcpp=Environment::namespace_env( "Rcpp" );
```

The Rcpp API : Lists for input / output

Actual code from the `earthmovdist` package on R-Forge

```
RcppExport SEXP emdL1(SEXP H1, SEXP H2, SEXP parms) {

  try {

    Rcpp::NumericVector h1(H1);       // double vector based on H1
    Rcpp::NumericVector h2(H2);       // double vector based on H2
    Rcpp::List rparam(parms);         // parameter from R based on parms
    bool verbose = Rcpp::as<bool>(rparam["verbose"]);

    [...]

    return Rcpp::NumericVector::create(Rcpp::Named("dist", d));

  } catch(std::exception &ex) {
     forward_exception_to_r(ex);
  } catch(...) {
     ::Rf_error("c++ exception (unknown reason)");
  }
  return R_NilValue;
}
```

The Rcpp API : example

```
SEXP foo( SEXP xs, SEXP ys ){
    Rcpp::NumericVector xx(xs), yy(ys) ;
    int n = xx.size() ;
    Rcpp::NumericVector res( n ) ;
    double x = 0.0, y = 0.0 ;
    for (int i=0; i<n; i++) {
        x = xx[i];
        y = yy[i];
        res[i] = (x < y) ? x*x : -(y*y);
    }
    return res ;
}
```

The Rcpp API : example

```
using namespace Rcpp ;
SEXP bar(){
    std::vector<double> z(10) ;
    List res = List::create(
      _["foo"] = NumericVector::create(1,2),
      _["bar"] = 3,
      _["bla"] = "yada yada",
      _["blo"] = z
      ) ;
    res.attr("class") = "myclass" ;
    return res ;
}
```

The Rcpp API : example

Inspired from a question on r-help

Faster code for `t(apply(x,1,cumsum))`

```
     [,1] [,2] [,3]               [,1] [,2] [,3]
[1,]    1    5    9         [1,]    1    6   15
[2,]    2    6   10    →    [2,]    2    8   18
[3,]    3    7   11         [3,]    3   10   21
[4,]    4    8   12         [4,]    4   12   24
```

The Rcpp API : example

Inspired from a question on r-help

Two R versions:

```
> # quite slow
> f.R1 <- function( x ){
+       t(apply(probs, 1, cumsum))
+ }
> # faster
> f.R2 <- function( x ){
+       y <- x
+       for( i in 2:ncol(x)){
+           y[,i] <- y[,i-1] + x[,i]
+       }
+       y
+ }
```

The Rcpp API : example

Inspired from a question on r-help

```
SEXP foo( SEXP x ){
  NumericMatrix input( x );

  // grab the number of rows and columns
  int nr = input.nrow(), nc = input.ncol();

  // create a new matrix to store the results
  NumericMatrix output = clone<NumericMatrix>(input);

  // edit the current column of the output using the previous
  // column and the current input column
  for( int i=1; i<nc; i++)
    output.column(i) =
        output.column(i-1) + input.column(i);

  return output;
}
```

The Rcpp API : example

Inspired from a question on r-help

version	elapsed time	relative
f.Rcpp	0.25	1.00
f.R2	0.46	1.87
f.R1	12.05	49.16

Using STL algorithms

C++ version of lapply using std::transform

```
src <- '
  Rcpp::List input(data);
  Rcpp::Function f(fun);
  Rcpp::List output(input.size());
  std::transform(
   input.begin(), input.end(),
   output.begin(),
   f );
  output.names() = input.names();
  return output;
  '
cpp_lapply <- cxxfunction(
  signature(data="list", fun = "function"),
  src, plugin="Rcpp")
```

Simple C++ version of lapply

Using the function

```
> cpp_lapply( faithful, summary )
$eruptions
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   1.60    2.16    4.00    3.49    4.45    5.10

$waiting
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   43.0    58.0    76.0    70.9    82.0    96.0
```

The Rcpp API : example

Calling an R function

From a project we are currently working on:

```
double evaluate(SEXP par_, SEXP fun_, SEXP rho_) {
   Rcpp::NumericVector par(par_);
   Rcpp::Function fun(fun_);
   Rcpp::Environment env(rho);

   Rcpp::Language funcall(fun, par);
   double res = Rcpp::as<double>(funcall.eval(env));

   return(res);
}
```

Yet using **Rcpp** here *repeatedly* as in function optimization is not yet competitive.

The Rcpp API : example

Calling an R function (plain API variant)

```
double evaluate(const double *param, SEXP par,
                           SEXP fcall, SEXP env) {
  // -- faster: direct access _assuming_ numeric vector
  memcpy(REAL(par), param, Rf_nrows(par) * sizeof(double));

  SEXP fn = ::Rf_lang2(fcall, par);    // could be done with Rcpp
  SEXP sexp_fvec = ::Rf_eval(fn, env);// but is slower right now

  double res = Rcpp::as<double>(sexp_fvec);
  return(res);
}
```

The Rcpp API : conversion from R to C++

Rcpp::as<T> handles conversion from SEXP to T.

```
template <typename T> T as( SEXP m_sexp)
    throw(not_compatible) ;
```

T can be:

- primitive type : int, double, bool, long, std::string
- any type that has a constructor taking a SEXP
- ... that specializes the as template
- ... that specializes the Exporter class template
- containers from the STL

more details in the Rcpp-extending vignette.

The Rcpp API : conversion from C++ to R

`Rcpp::wrap<T>` handles conversion from `T` to `SEXP`.

```
template <typename T>
SEXP wrap( const T& object ) ;
```

`T` can be:

- primitive type : `int`, `double`, `bool`, `long`, `std::string`
- any type that has a an operator `SEXP`
- ... that specializes the `wrap` template
- ... that has a nested type called `iterator` and member functions `begin` and `end`
- containers from the STL `vector<T>`, `list<T>`, `map<string,T>`, etc ... (where `T` is itself wrappable)

more details in the `Rcpp-extending` vignette.

The Rcpp API : conversion examples

```
typedef std::vector<double> Vec ;
int x_ = as<int>( x ) ;
double y_ = as<double>( y_ ) ;
VEC z_ = as<VEC>( z_ ) ;

wrap( 1 ) ;       // INTSXP
wrap( "foo" ) ; // STRSXP

typedef std::map<std::string,Vec> Map ;
Map foo( 10 ) ;
Vec f1(4) ;
Vec f2(10) ;
foo.insert( "x", f1 ) ;
foo.insert( "y", f2 ) ;
wrap( foo ) ; // named list of numeric vectors
```

The Rcpp API : *implicit* conversion examples

```
Environment env = ... ;
List list = ... ;
Function rnorm( "rnorm") ;

// implicit calls to as
int x = env["x"] ;
double y = list["y"];

// implicit calls to wrap
rnorm( 100, _["mean"] = 10 ) ;
env["x"] = 3;
env["y"] = "foo" ;
List::create( 1, "foo", 10.0, false ) ;
```

inline

The inline package

inline by Oleg Sklyar *et al* is a wonderfully useful little package.

We extended it to work with **Rcpp** (and related packages such as **RcppArmadillo**, see below).

```
# default plugin
fx <- cxxfunction(signature(x = "integer", y = "numeric") ,
                  'return ScalarReal( INTEGER(x)[0]
                          * REAL(y)[0] ); ')
fx( 2L, 5 )

# Rcpp plugin
fx <- cxxfunction(signature(x = "integer", y = "numeric"),
                  'return wrap(as<int>(x)
                          * as<double>(y));',
                  plugin = "Rcpp" )
fx( 2L, 5 )
```

Compiles, links and loads C, C++ and Fortran.

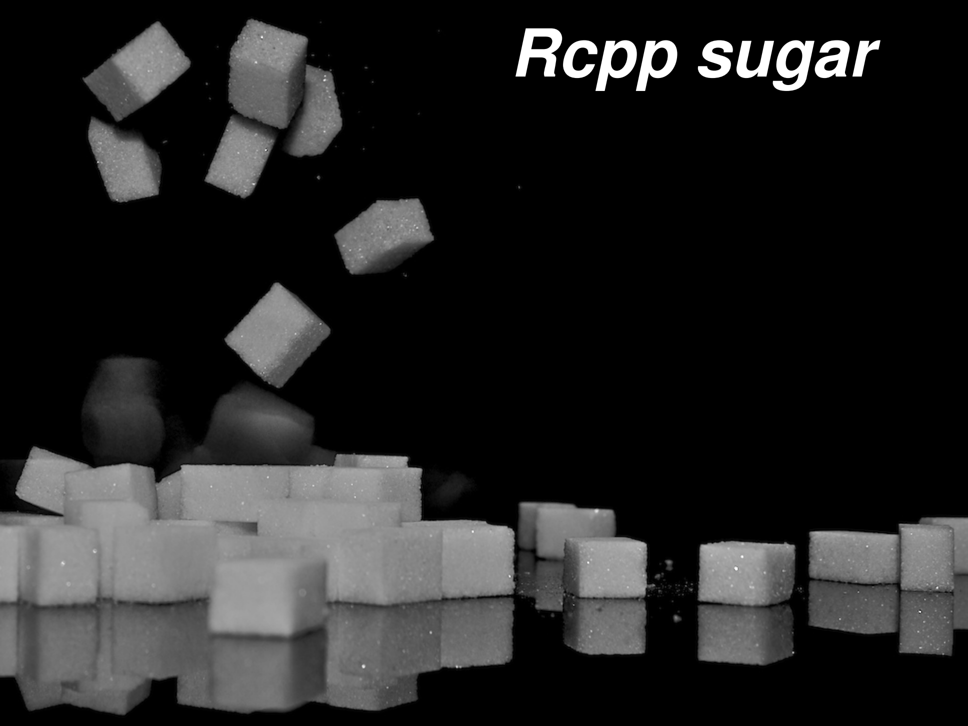
The inline package

Also works for templated code – *cf* Whit on rcpp-devel last month

```
inc <- ’
#include <iostream>
#include <armadillo>
#include <cppbugs/cppbugs.hpp>

using namespace arma;
using namespace cppbugs;

class TestModel: public MCModel {
public:
  const mat& y; // given
  const mat& X; // given
  Normal<vec> b;
  Uniform<double> tau_y;
  Deterministic<mat> y_hat;
  Normal<mat> likelihood;
  Deterministic<double> rsq;

  TestModel(const mat& y_,const mat& X_):
        y(y_), X(X_), b(randn<vec>(X_.n_cols)), tau_y(1),
        y_hat(X*b.value), likelihood(y_,true), rsq(0) {
   [...]
’
```

`inc=` includes headers before the `body=` — and the templated `CppBUGS` package by Whit now outperforms PyMC / Bugs.

Rcpp sugar



Sugar : motivation

```
int n = x.size() ;
 NumericVector res1( n ) ;
 double x_ = 0.0, y_ = 0.0 ;
 for( int i=0; i<n; i++){
        x_ = x[i] ;y_ = y[i] ;
        if( R_IsNA(x_) || R_IsNA(y_) ){
            res1[i] = NA_REAL;
        } else if( x_ < y_ ){
            res1[i] = x_ * x_ ;
        } else {
            res1[i] = -( y_ * y_)  ;
        }
    }
```

Sugar : motivation

We missed the R syntax :

```
> ifelse( x < y, x*x, -(y*y) )
```

`sugar` brings it into C++

```
SEXP foo( SEXP xx, SEXP yy){
    NumericVector x(xx), y(yy) ;
    return ifelse( x < y, x*x, -(y*y) ) ;
}
```

Sugar : another example

```
double square( double x){
  return x*x ;
}

SEXP foo( SEXP xx ){
  NumericVector x(xx) ;
  return sapply( x, square ) ;
}
```

Sugar : contents

- logical operators: `<`, `>`, `<=`, `>=`, `==`, `!=`
- arithmetic operators: `+`, `-`, `*`, `/`
- functions on vectors: `abs`, `all`, `any`, `ceiling`, `diag`, `diff`, `exp`, `head`, `ifelse`, `is_na`, `lapply`, `pmin`, `pmax`, `pow`, `rep`, `rep_each`, `rep_len`, `rev`, `sapply`, `seq_along`, `seq_len`, `sign`, `tail`
- functions on matrices: `outer`, `col`, `row`, `lower_tri`, `upper_tri`, `diag`
- statistical functions (dpqr) : `rnorm`, `dpois`, `qlogis`, etc ...

More information in the `Rcpp-sugar` vignette.

Sugar : benchmarks

expression	sugar	R	R / sugar
`any(x*y<0)`	0.000447	4.86	10867
`ifelse(x<y,x*x,-(y*y))`	1.331	22.29	16.74
`ifelse(x<y,x*x,-(y*y))` (⋆)	0.832	21.59	24.19
`sapply(x,square)`	0.240	138.71	577.39

Benchmarks performed on OSX SL / R 2.12.0 alpha (64 bit) on a MacBook Pro (i5).

⋆ : version includes optimization related to the absence of missing values

Sugar : benchmarks

Benchmarks of the convolution example from Writing R Extensions.

Implementation	Time in millisec	Relative to R API
R API (as benchmark)	218	
Rcpp sugar	145	0.67
`NumericVector::iterator`	217	1.00
`NumericVector::operator[]`	282	1.29
`RcppVector<double>`	683	3.13

Table: Convolution of *x* and *y* (200 values), repeated 5000 times.

Extract from the article *Rcpp: Seamless R and C++ integration*, accepted for publication in the R Journal.

RInside

From RApache to littler to RInside

See the file `RInside/standard/rinside_sample0.cpp`

Jeff Horner’s work on `RApache` lead to joint work in `littler`, a scripting / cmdline front-end. As it embeds R and simply ’feeds’ the REPL loop, the next step was to embed R in proper C++ classes: `RInside`.

```
#include <RInside.h>                    // for the embedded R via RInside

int main(int argc, char *argv[]) {

  RInside R(argc, argv);                // create an embedded R instance

  R["txt"] = "Hello, world!\n";         // assign a char* (string) to ’txt’

  R.parseEvalQ("cat(txt)");             // eval init string, ignore any returns

  exit(0);
}
```

Another simple example

See `RInside/standard/rinside_sample8.cpp` (in SVN, older version in pkg)

This shows some of the assignment and converter code:

```
#include <RInside.h>                    // for the embedded R via RInside

int main(int argc, char *argv[]) {

    RInside R(argc, argv);              // create an embedded R instance

    R["x"] = 10 ;
    R["y"] = 20 ;

    R.parseEvalQ("z <- x + y") ;

    int sum = R["z"];

    std::cout << "10 + 20 = " << sum << std::endl ;
    exit(0);
}
```

A finance example

See the file `RInside/standard/rinside_sample4.cpp` (edited)

```
#include <RInside.h>                    // for the embedded R via RInside
#include <iomanip>
int main(int argc, char *argv[]) {
    RInside R(argc, argv);              // create an embedded R instance
    SEXP ans;
    R.parseEvalQ("suppressMessages(library(fPortfolio))");
    txt = "lppData <- 100 * LPP2005.RET[, 1:6]; "
          "ewSpec <- portfolioSpec(); nAssets <- ncol(lppData); ";
    R.parseEval(txt, ans);              // prepare problem
    const double dvec[6] = { 0.1, 0.1, 0.1, 0.1, 0.3, 0.3 }; // weights
    const std::vector<double> w(dvec, &dvec[6]);
    R.assign( w, "weightsvec");         // assign STL vec to Rs weightsvec

    R.parseEvalQ("setWeights(ewSpec) <- weightsvec");
    txt = "ewPortfolio <- feasiblePortfolio(data = lppData, spec = ewSpec, "
          "                          constraints = \"LongOnly\"); "
          "print(ewPortfolio); "
          "vec <- getCovRiskBudgets(ewPortfolio@portfolio)";
    ans = R.parseEval(txt);             // assign covRiskBudget weights to ans
    Rcpp::NumericVector V(ans);         // convert SEXP variable to an RcppVector

    ans = R.parseEval("names(vec)");            // assign columns names to ans
    Rcpp::CharacterVector n(ans);

    for (int i=0; i<names.size(); i++) {
      std::cout << std::setw(16) << n[i] << "\t" << std::setw(11) << V[i] << "\n";
    }
    exit(0);
}
```

RInside and C++ integration

See the file `RInside/standard/rinside_sample9.cpp`

```
#include <RInside.h>                    // for the embedded R via RInside

// a c++ function we wish to expose to R
const char* hello( std::string who ){
        std::string result( "hello " ) ;
        result += who ;
        return result.c_str() ;
}

int main(int argc, char *argv[]) {

    // create an embedded R instance
    RInside R(argc, argv);

    // expose the "hello" function in the global environment
    R["hello"] = Rcpp::InternalFunction( &hello ) ;

    // call it and display the result
    std::string result = R.parseEval("hello(’world’)") ;
    std::cout << "hello( ’world’) =  " << result << std::endl ;

    exit(0);
}
```

And another *parallel* example

See the file `RInside/mpi/rinside_mpi_sample2.cpp`

```cpp
// MPI C++ API version of file contributed by Jianping Hua

#include <mpi.h>     // mpi header
#include <RInside.h> // for the embedded R via RInside

int main(int argc, char *argv[]) {

  MPI::Init(argc, argv);                      // mpi initialization
  int myrank = MPI::COMM_WORLD.Get_rank();    // obtain current node rank
  int nodesize = MPI::COMM_WORLD.Get_size();  // obtain total nodes running.

  RInside R(argc, argv);                      // create an embedded R instance

  std::stringstream txt;
  txt << "Hello from node " << myrank        // node information
      << " of " << nodesize << " nodes!" << std::endl;
  R.assign( txt.str(), "txt");                // assign string to R variable txt

  std::string evalstr = "cat(txt)";           // show node information
  R.parseEvalQ(evalstr);                      // eval the string, ign. any returns

  MPI::Finalize();                            // mpi finalization

  exit(0);
}
```

RInside workflow

- C++ programs compute, gather or aggregate raw data.
- Data is saved and analysed before a new ’run’ is launched.
- With `RInside` we now skip a step:
  - collect data in a vector or matrix
  - pass data to R — easy thanks to `Rcpp` wrappers
  - pass one or more short ’scripts’ as strings to R to evaluate
  - pass data back to C++ programm — easy thanks to `Rcpp` converters
  - resume main execution based on new results
- A number of simple examples ship with `RInside`
  - *nine* different examples in `examples/standard`
  - *four* different examples in `examples/mpi`

RProtoBuf

About Google ProtoBuf

Quoting from the page at Google Code:

> *Protocol buffers are a flexible, efficient, automated mechanism for serializing structured data—think XML, but smaller, faster, and simpler.*
>
> *You define how you want your data to be structured once, then you can use special generated source code to easily write and read your structured data to and from a variety of data streams and using a variety of languages.*
>
> *You can even update your data structure without breaking deployed programs that are compiled against the "old" format.*

Google provides native bindings for C++, Java and Python.

Example from the protobuf page

Create/update a message

```
message Person {
  required int32 id = 1;
  required string name = 2;
  optional string email = 3;
}
```

C++

```
Person person;
person.set_id(123);
person.set_name("Bob");
person.set_email("bob@example.com");

fstream out("person.pb", ios::out |
ios::binary | ios::trunc);
person.SerializeToOstream(&out);
out.close();
```

R/RProtoBuf

```
> library( RProtoBuf )
> ## create Bob
> bob <- new( tutorial.Person)
> ## assign to components
> bob$id <- 123
> bob$name <- "Bob"
> bob$email <- "bob@example.com"
> ## and write out
> serialize( bob, "person.pb" )
```

Example from the protobuf page

Reading from a file and access content of the message

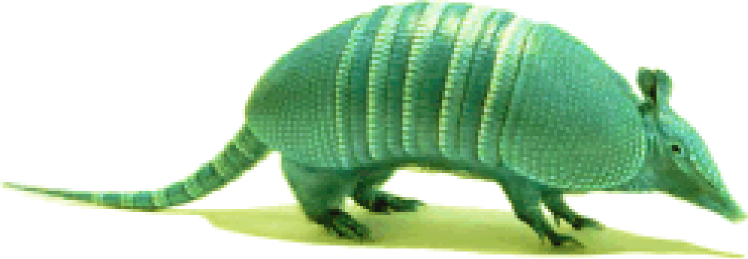
C++

```
Person person;
fstream in("person.pb", ios::in |
ios::binary);
if (!person.ParseFromIstream(&in)) {
  cerr << "Failed to parse person.pb." <<
endl;
  exit(1);
}

cout << "ID: " << person.id() << endl;
cout << "name: " << person.name() << endl;
if (person.has_email()) {
  cout << "e-mail: " << person.email() <<
endl;
}
```

R/RProtoBuf

```
> person <- read(tutorial.Person, "person.pb")
> cat( "ID: ", person$id, "\n" )
> cat( "name: ", person$name, "\n" )
> if( person$has( "email" ) ){
+     cat( "email: ", person$email, "\n" )
+ }
```

RcppArmadillo



Linear regression via Armadillo: lmArmadillo example

Also see the directory `Rcpp/examples/FastLM`

```
lmArmadillo <- function() {
  src <- '
  Rcpp::NumericVector yr(Ysexp);
  Rcpp::NumericVector Xr(Xsexp);          // actually an n x k matrix
  std::vector<int> dims = Xr.attr("dim");
  int n = dims[0], k = dims[1];
  arma::mat X(Xr.begin(), n, k, false);   // use advanced armadillo constructors
  arma::colvec y(yr.begin(), yr.size());
  arma::colvec coef = solve(X, y);        // model fit
  arma::colvec resid = y - X*coef;        // comp. std.errr of the coefficients
  arma::mat covmat = trans(resid)*resid/(n-k) * arma::inv(arma::trans(X)*X);

  Rcpp::NumericVector coefr(k), stderrestr(k);
  for (int i=0; i<k; i++) {               // with RcppArmadillo templ. conv.
      coefr[i]      = coef[i];            // this would not be needed but we only
      stderrestr[i] = sqrt(covmat(i,i));  // assume Rcpp.h here
  }
  return Rcpp::List::create(Rcpp::Named( "coefficients", coefr),
                            Rcpp::Named( "stderr", stderrestr));
  '
  ## turn into a function that R can call
  fun <- cppfunction(signature(Ysexp="numeric", Xsexp="numeric"),
                     src, plugin="RcppArmadillo")
}
```

Linear regression via Armadillo: RcppArmadillo

See `fastLm` in the RcppArmadillo package

`fastLm` in the new `RcppArmadillo` release does even better:

```
#include <RcppArmadillo.h>
extern "C" SEXP fastLm(SEXP ys, SEXP Xs) {
  try {
    Rcpp::NumericVector yr(ys);                 // creates Rcpp vector from SEXP
    Rcpp::NumericMatrix Xr(Xs);                 // creates Rcpp matrix from SEXP
    int n = Xr.nrow(), k = Xr.ncol();
    arma::mat X(Xr.begin(), n, k, false);      // reuses memory and avoids extra copy
    arma::colvec y(yr.begin(), yr.size(), false);
    arma::colvec coef = arma::solve(X, y);      // fit model y ~ X
    arma::colvec res = y - X*coef;              // residuals

    double s2 =
        std::inner_product(res.begin(),res.end(),res.begin(),double())/(n-k);
                                                // std.errors of coefficients
    arma::colvec stderr =
        arma::sqrt(s2*arma::diagvec(arma::inv(arma::trans(X)*X)));

    return Rcpp::List::create(Rcpp::Named("coefficients") = coef,
                              Rcpp::Named("stderr")       = stderr,
                              Rcpp::Named("df")           = n - k);
  } catch( std::exception &ex ) {
    forward_exception_to_r( ex );
  } catch(...) {
    ::Rf_error( "c++ exception (unknown reason)" );
  }
  return R_NilValue; // -Wall
}
```

Linear regression via GNU GSL: RcppGSL

See `fastLm` in the RcppGSL package (on R-Forge)

```
#include <RcppArmadillo.h>
extern "C" SEXP fastLm(SEXP ys, SEXP Xs) {
  BEGIN_RCPP
  RcppGSL::vector<double> y = ys;       // create gsl data structures from SEXP
  RcppGSL::matrix<double> X = Xs;
  int n = X.nrow(), k = X.ncol();
  double chisq;
  RcppGSL::vector<double> coef(k);      // to hold the coefficient vector
  RcppGSL::matrix<double> cov(k,k);     // and the covariance matrix
  // the actual fit requires working memory we allocate and free
  gsl_multifit_linear_workspace *work = gsl_multifit_linear_alloc (n, k);
  gsl_multifit_linear (X, y, coef, cov, &chisq, work);
  gsl_multifit_linear_free (work);
  // extract the diagonal as a vector view
  gsl_vector_view diag = gsl_matrix_diagonal(cov) ;
  // currently there is not a more direct interface in Rcpp::NumericVector
  // that takes advantage of wrap, so we have to do it in two steps
  Rcpp::NumericVector stderr ; stderr = diag;
  std::transform( stderr.begin(), stderr.end(), stderr.begin(), sqrt );
  Rcpp::List res = Rcpp::List::create(Rcpp::Named("coefficients") = coef,
                                      Rcpp::Named("stderr") = stderr,
                                      Rcpp::Named("df") = n - k);
  // free all the GSL vectors and matrices -- as these are really C data structures
  // we cannot take advantage of automatic memory management
  coef.free(); cov.free(); y.free(); X.free();
  return res;       // return the result list to R
  END_RCPP
}
```

Objects

Lexical Scoping

S3 classes

S4 classes

Reference (R5) classes

C++ classes

Protocol Buffers

Fil rouge: bank account example

- Data:
  - The balance
  - Authorized overdraft

- Operations:
  - Open an account
  - Get the balance
  - Deposit
  - Withdraw

Lexcial Scoping

```
> open.account <- function(total, overdraft = 0.0){
+     deposit <- function(amount) {
+         if( amount < 0 )
+             stop( "deposits must be positive" )
+         total <<- total + amount
+     }
+     withdraw <- function(amount) {
+         if( amount < 0 )
+             stop( "withdrawals must be positive" )
+         if( total - amount < overdraft )
+             stop( "you cannot withdraw that much" )
+         total <<- total - amount
+     }
+     balance <- function() {
+         total
+     }
+     list( deposit = deposit, withdraw = withdraw,
+         balance = balance )
+ }
> romain <- open.account(500)
> romain$balance()
[1] 500

> romain$deposit(100)
> romain$withdraw(200)
> romain$balance()
[1] 400
```

S3 classes

- Any R object with a **class** attribute
- Very easy
- Very dangerous
- Behaviour is added through S3 generic functions

```
> Account <- function( total, overdraft = 0.0 ){
+     out <- list( balance = total, overdraft = overdraft )
+     class( out ) <- "Account"
+     out
+ }
> balance <- function(x){
+     UseMethod( "balance" )
+ }
> balance.Account <- function(x) x$balance
```

S3 classes

```
> deposit <- function(x, amount){
+     UseMethod( "deposit" )
+ }
> deposit.Account <- function(x, amount) {
+     if( amount < 0 )
+         stop( "deposits must be positive" )
+     x$balance <- x$balance + amount
+     x
+ }
> withdraw <- function(x, amount){
+     UseMethod( "withdraw" )
+ }
> withdraw.Account <- function(x, amount) {
+     if( amount < 0 )
+         stop( "withdrawals must be positive" )
+     if( x$balance - amount < x$overdraft )
+         stop( "you cannot withdraw that much" )
+     x$balance <- x$balance - amount
+     x
+ }
```

S3 classes

Example use:

```
> romain <- Account( 500 )
> balance( romain )
[1] 500

> romain <- deposit( romain, 100 )
> romain <- withdraw( romain, 200 )
> balance( romain )
[1] 400
```

S4 classes

- Formal class definition
- Validity checking
- Formal generic functions and methods
- Very verbose, both in code and documentation

S4 classes

```
> setClass( "Account",
+     representation(
+         balance = "numeric",
+         overdraft = "numeric"
+     ),
+     prototype = prototype(
+         balance = 0.0,
+         overdraft = 0.0
+     ),
+     validity = function(object){
+         object@balance > object@overdraft
+     }
+ )
[1] "Account"

> setGeneric( "balance",
+     function(x) standardGeneric( "balance" )
+ )
[1] "balance"

> setMethod( "balance", "Account",
+     function(x) x@balance
+ )
[1] "balance"
```

S4 classes

```
> setGeneric( "deposit",
+     function(x, amount) standardGeneric( "deposit" )
+ )
[1] "deposit"

> setMethod( "deposit",
+     signature( x = "Account", amount = "numeric" ),
+     function(x, amount){
+         new( "Account" ,
+             balance = x@balance + amount,
+             overdraft = x@overdraft
+         )
+     }
+ )
[1] "deposit"
```

S4 classes

```
> romain <- new( "Account", balance = 500 )
> balance( romain )
[1] 500

> romain <- deposit( romain, 100 )
> romain <- withdraw( romain, 200 )
> balance( romain )
[1] 400
```

Reference (R5) classes

- Real S4 classes: formalism, dispatch, ...
- Passed by Reference
- Easy to use

Reference (R5) classes

```
> Account <- setRefClass( "Account_R5",
+     fields = list(
+         balance = "numeric",
+         overdraft = "numeric"
+     ),
+     methods = list(
+         withdraw = function( amount ){
+             if( amount < 0 )
+                 stop( "withdrawal must be positive" )
+             if( balance - amount < overdraft )
+                 stop( "overdrawn" )
+             balance <<- balance - amount
+         },
+         deposit = function(amount){
+             if( amount < 0 )
+                 stop( "deposits must be positive" )
+             balance <<- balance + amount
+         }
+     )
+ )
> x <- Account$new( balance = 10.0, overdraft = 0.0 )
> x$withdraw( 5 )
> x$deposit( 10 )
> x$balance
[1] 15
```

Reference (R5) classes

Real *pass by reference* :

```
> borrow <- function( x, y, amount = 0.0 ){
+     x$withdraw( amount )
+     y$deposit( amount )
+     invisible(NULL)
+ }
> romain <- Account$new( balance = 5000, overdraft = 0.0 )
> dirk <- Account$new( balance = 3, overdraft = 0.0 )
> borrow( romain, dirk, 2000 )
> romain$balance
[1] 3000

> dirk$balance
[1] 2003
```

Reference (R5) classes

Adding a method dynamically to a class :

```
> Account$methods(
+     borrow = function(other, amount){
+         deposit( amount )
+         other$withdraw( amount )
+         invisible(NULL)
+     }
+ )
> romain <- Account$new( balance = 5000, overdraft = 0.0 )
> dirk <- Account$new( balance = 3, overdraft = 0.0 )
> dirk$borrow( romain, 2000 )
> romain$balance
[1] 3000

> dirk$balance
[1] 2003
```

C++ classes

```
class Account {
public:
    Account() : balance(0.0), overdraft(0.0){}

    void withdraw( double amount ){
        if( balance - amount < overdraft )
            throw std::range_error( "no way" ) ;
        balance -= amount ;
    }

    void deposit( double amount ){
        balance += amount ;
    }

    double balance ;

private:
    double overdraft ;
} ;
```

C++ classes

Exposing to R through Rcpp modules:

```
RCPP_MODULE(yada){
    class_<Account>( "Account" )

        // expose the field
        .field_readonly( "balance", &Account::balance )

        // expose the methods
        .method( "withdraw", &Account::withdraw )
        .method( "deposit", &Account::deposit ) ;

}
```

Use it in R:

```
> Account <- yada$Account
> romain <- Account$new()
> romain$deposit( 10 )
> romain$withdraw( 2 )
> romain$balance
[1] 8
```

Protocol Buffers

Define the message type, in `Account.proto` :

```
package foo ;

message Account {
    required double balance = 1 ;
    required double overdraft = 2 ;
}
```

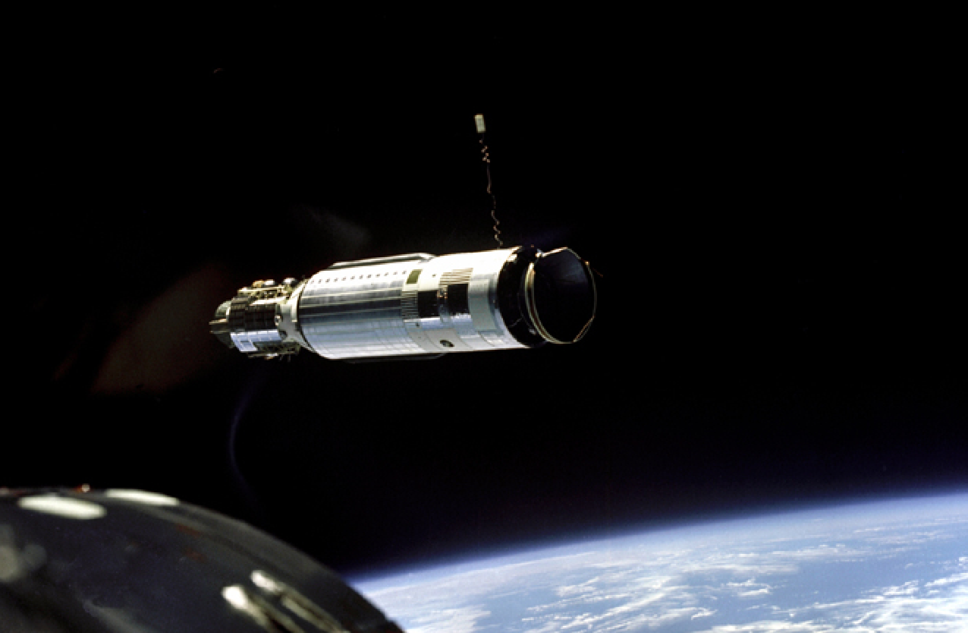
Load it into R with `RProtoBuf`:

```
> require( RProtoBuf )
> loadProtoFile( "Account.proto" )
```

Use it:

```
> romain <- new( foo.Account,
+     balance = 500, overdraft = 10 )
> romain$balance
```

Rcpp modules



Modules: expose C++ to R

```
const char* hello( const std::string& who ){
    std::string result( "hello " ) ;
    result += who ;
    return result.c_str() ;
}

RCPP_MODULE(yada){
    using namespace Rcpp ;
    function( "hello", &hello ) ;
}
```

```
> yada <- Module( "yada" )
> yada$hello( "world" )
```

Modules: expose C++ classes to R

```
class World {
public:
    World() : msg("hello"){}
    void set(std::string msg) {
        this->msg = msg;
    }
    std::string greet() {
        return msg;
    }
private:
    std::string msg;
};

void clearWorld( World* w){
    w->set( "" ) ;
}
```

Modules: expose C++ classes to R

C++ side: declare *what* to expose

```
RCPP_MODULE(yada){
    using namespace Rcpp ;

    class_<World>( "World" )
        .method( "greet", &World::greet )
        .method( "set", &World::set )
        .method( "clear", &clearWorld )
    ;

}
```

Modules: on the R side

R side: based on R 2.12.0 reference classes (aka R5), see `?ReferenceClasses`

```
> World <- yada$World
> w <- new( World )
> w$greet()
[1] "hello"

> w$set( "hello world")
> w$greet()
[1] "hello world"

> w$clear()
> w$greet()
[1] ""
```

Creating a package using Rcpp

A simple yet reliable strategy is to

- prototype code using **inline**
- call `package.skeleton` the resulting function generated by `cxxfunction` — and magic ensues
- Kidding aside, **inline** provides a variant of `package.skeleton` that knows how to employ the information in the generated function.

Creating a package using Rcpp

```
foo <- cxxfunction(list(tic=signature(x="numeric",y="numeric"),
                        tac=signature(x="numeric",y="numeric")),
                   list(tic="return Rcpp::wrap( sqrt(pow(Rcpp::as<double>(x), 2) +
                                      pow(Rcpp::as<double>(y), 2)));",
                        tac="return Rcpp::wrap( sqrt(fabs(Rcpp::as<double>(x)) +
                                      fabs(Rcpp::as<double>(y))));"),
                   plugin="Rcpp")

foo$tic(-2,  3)
foo$tac( 2, -3)

package.skeleton("myPackage", foo)
```

Further Reading

Rcpp comes with eight vignettes:

- Rcpp-introduction: A overview article covering the core features
- Rcpp-FAQ: Answers to (in)frequently asked questions
- Rcpp-package: How to use Rcpp in your own package
- Rcpp-extensions: How to extend Rcpp as RcppArmadillo or RcppGSL do
- Rcpp-sugar: An overview of ’Rcpp sugar’
- Rcpp-modules: An overview of ’Rcpp modules
- Rcpp-quickref: A quick reference guide to the Rcpp API
- Rcpp-unittest: Autogenerated results from running 700+ unit tests

Further Reading

The unit tests also provide usage examples.

CRAN now lists fifteen packages depending on **Rcpp** – these also provide working examples.

The `rcpp-devel` mailing list (and its archive) is a further resource.

Want to learn more ?

- Check the vignettes
- Questions on the `Rcpp-devel` mailing list
- Hands-on training courses
- Commercial support

Romain François **romain@r-enthusiasts.com**
Dirk Eddelbuettel **edd@debian.org**