R and C++: Seamless Integration using Rcpp

Dr. Dirk Eddelbuettel
edd@debian.org
dirk.eddelbuettel@R-Project.org
dirk@eddelbuettel.com

Joint work with Romain François

Seattle, WA
December 7, 2011

R and C++: Why, How, What

The three main questions for this talk:

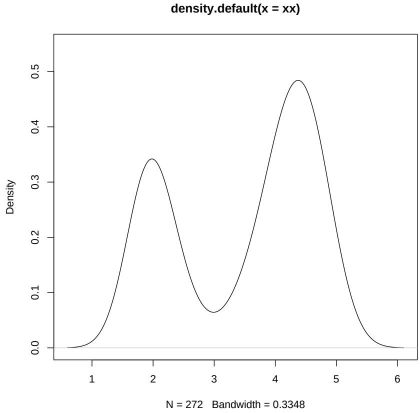
- Why? There are several reasons discussed next ...
- How? We will show some simple illustrations ...
- What? This will also be covered ...

Outline

Why R? – A Simple Example

Courtesy of Greg Snow via r-help during Sep 2010

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

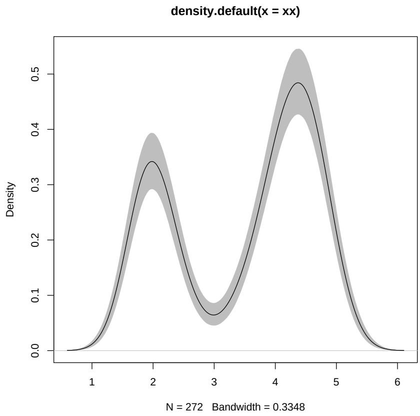


Standard R use: load some data, estimate a density, plot it.

Why R? – A Simple Example, extended

Now with a simulation-based estimation uncertainty band for the nonparametric density.

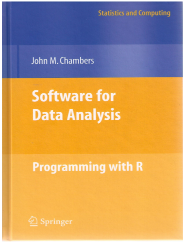
```
xx <- faithful$eruptions
fit1 <- density(xx)
fit2 <- replicate(10000, {
  x <- sample(xx,replace=TRUE);
  density(x, from=min(fit1$x),
          to=max(fit1$x))$y
})
fit3 <- apply(fit2, 1,
  quantile,c(0.025,0.975))
plot(fit1, ylim=range(fit3))
polygon(c(fit1$x,rev(fit1$x)),
  c(fit3[1,], rev(fit3[2,])),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



What other language can do that in seven statements?

Motivation

Why would extending R via C/C++/Rcpp be of interest?



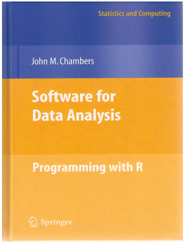
Chambers. *Software for Data Analysis: Programming with R*. Springer, 2008

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

Why would extending R via C/C++/Rcpp be of interest?



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

Why would extending R via C/C++/Rcpp be of interest?

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

So the why...

The *why* boils down to:

- **speed!** Often a good enough reason for us ... and a major focus for us today.
- **new things!** We can bind to libraries and tools that would otherwise be unavailable
- **references!** Chambers quote from 2008 somehow foreshadowed the work on *Reference Classes* released with R 2.12 and which work very well with **Rcpp** modules. More generally, we can do pass-by-reference in C/C++.

Why extend with C++?

That's a near religious question.

- C is a plausible choice as R is written in it – but too bare.
- C++ is close to C, but “more”. Paraphrasing Meyers, we can call it a language with “four different paradigms inside”.
- C++ may be intimidating. It shouldn't be. C++ in 2011 is very different from C++ in 1991.
- C++ is industrial strength. Many excellent libraries. Great support for scientific computing. Many APIs.
- Let's focus on *Extending R, and taking C++ as a given*.
- **Rcpp** lets you extend R in the easiest possible way. C++ is just a tool in that context.

Outline

R Extension Basics

Let's recap what the "Writing R Extensions" manual says:

- The primary interface is the `.Call()` function
- It can take a variable number of `SEXP` variables on input.
- It returns a single `SEXP`.
- So *everything* revolves around `SEXP` object.
- But ... what exactly is a `SEXP`?

SEXP: Opaque Pointer to S Expression (SEXPREC)

- The gory details are in Section 1.1 “SEXPs” of the *R Internals* manual
- SEXPs are opaque pointers, and several distinct types are aggregated in a C union type
- Section 1.1.1 “SEXPTYPE” lists the 26 different types a SEXP could point to
- It’s a mess, but it is the best you can do if C is all you have.
- There are macros systems (two unfortunately) to help shield the innards of SEXPs.

Comparing the R API to Rcpp: Vectors

Using the basic C API for R.

```
#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];
    xab[i] = (x < y) ? x*x : -(y*y);
  }
  UNPROTECT(3);
  return(ab);
}
```

Comparing the R API to Rcpp: Vectors

Using the basic C API for R.

```
#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];
    xab[i] = (x < y) ? x*x : -(y*y);
  }
  UNPROTECT(3);
  return(ab);
}
```

Need `PROTECT` and `UNPROTECT`, multiple explicit casts, and pre-scrub results vector: Tedious!

Comparing the R API to Rcpp: Vectors

Using the basic C API for R.

```
#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];
    xab[i] = (x < y) ? x*x : -(y*y);
  }
  UNPROTECT(3);
  return(ab);
}
```

Need `PROTECT` and `UNPROTECT`, multiple explicit casts, and pre-scrub results vector: Tedious!

Or using **Rcpp**.

```
#include <Rcpp.h>
extern "C" SEXP v2(SEXP a, SEXP b) {
  NumericVector x(a), y(b);
  int n = x.size();
  NumericVector res(n);
  for (int i=0; i<n; i++) {
    res[i] = (x[i] < y[i]) ?
              x[i]*x[i] : -(y[i]*y[i]);
  }
  return res;
}
```

Comparing the R API to Rcpp: Vectors

Using the basic C API for R.

```
#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];
    xab[i] = (x < y) ? x*x : -(y*y);
  }
  UNPROTECT(3);
  return(ab);
}
```

Need `PROTECT` and `UNPROTECT`, multiple explicit casts, and pre-scrub results vector: Tedious!

Or using **Rcpp**.

```
#include <Rcpp.h>
extern "C" SEXP v2(SEXP a, SEXP b) {
  NumericVector x(a), y(b);
  int n = x.size();
  NumericVector res(n);
  for (int i=0; i<n; i++) {
    res[i] = (x[i] < y[i]) ?
               x[i]*x[i] : -(y[i]*y[i]);
  }
  return res;
}
```

or using **Rcpp** sugar:

```
#include <Rcpp.h>
extern "C" SEXP v2(SEXP a, SEXP b) {
  NumericVector x(a), y(b);
  NumericVector res =
    ifelse(x < y, x*x, -(y*y));
  return res;
}
```

Comparing the R API to Rcpp: Vectors

Using the basic C API for R.

```
#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];
    xab[i] = (x < y) ? x*x : -(y*y);
  }
  UNPROTECT(3);
  return(ab);
}
```

Need `PROTECT` and `UNPROTECT`, multiple explicit casts, and pre-scrub results vector: Tedious!

Or using **Rcpp**.

```
#include <Rcpp.h>
extern "C" SEXP v2(SEXP a, SEXP b) {
  NumericVector x(a), y(b);
  int n = x.size();
  NumericVector res(n);
  for (int i=0; i<n; i++) {
    res[i] = (x[i] < y[i]) ?
             x[i]*x[i] : -(y[i]*y[i]);
  }
  return res;
}
```

or using **Rcpp** sugar:

```
#include <Rcpp.h>
extern "C" SEXP v2(SEXP a, SEXP b) {
  NumericVector x(a), y(b);
  NumericVector res =
    ifelse(x < y, x*x, -(y*y));
  return res;
}
```

In R, for comparison:

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

Comparing the R API to Rcpp: Vectors – R use

With magic provided by the 'inline' package (Sklyar et al)

```
R> ex1c <- cfunction(signature(a="numeric",
+                              b="numeric"),
+                    body='
+   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];
+     xab[i] = (x < y) ? x*x : -(y*y);
+   }
+   UNPROTECT(3);
+   return(ab);
+ ')
R> a <- c(1,2,3,4)
R> b <- c(4,1,4,1)
```

```
R> ex1rcpp <-
+     cxxfunction(signature(a="numeric",
+                           b="numeric"),
+                 plugin="Rcpp", body='
+   NumericVector x(a), y(b);
+   int n = x.size();
+   NumericVector res(n);
+   for (int i=0; i<n; i++) {
+     res[i] = (x[i] < y[i]) ?
+              x[i]*x[i] :
+             -(y[i]*y[i]);
+   }
+   return res;
+ ')
R> stopifnot(all.equal(ex1c(a,b),
+                      ex1rcpp(a,b)))
R> ex1rcppSugar <-
+     cxxfunction(signature(a="numeric",
+                           b="numeric"),
+                 plugin="Rcpp", body='
+   NumericVector x(a), y(b);
+   NumericVector res =
+          ifelse(x < y, x*x, -(y*y));
+   return res;
+ ')
R> stopifnot(all.equal(ex1c(a,b),
+                      ex1rcppSugar(a,b)))
R>
```

Comparing the R API to Rcpp: Char Vectors

Using the basic C API for R.

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

Comparing the R API to Rcpp: Char Vectors

Using the basic C API for R.

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

Need to remember to

- use `STRSXP`,
- allocate vectors,
- set elements as string elements (different from basic vectors).

Comparing the R API to Rcpp: Char Vectors

Using the basic C API for R.

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

Or using **Rcpp**.

```
#include <Rcpp.h>
extern "C" SEXP foobarRcpp(){
  StringVector res(2);
  res[0] = "foo";
  res[1] = "bar";
  return res ;
}
```

Need to remember to

- use `STRSXP`,
- allocate vectors,
- set elements as string elements (different from basic vectors).

Comparing the R API to Rcpp: Char Vectors

Using the basic C API for R.

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

Need to remember to

- use `STRSXP`,
- allocate vectors,
- set elements as string elements (different from basic vectors).

Or using **Rcpp**.

```
#include <Rcpp.h>
extern "C" SEXP foobarRcpp(){
  StringVector res(2);
  res[0] = "foo";
  res[1] = "bar";
  return res ;
}
```

Or using R:

```
res <- c("foo", "bar")
```

Comparing the R API to Rcpp: Functions

Using the basic C API for R.

```
#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)
     )
   )
  ));
  GetRNGState();
  SEXP res = PROTECT(eval(call,R_GlobalEnv));
  PutRNGState();
  UNPROTECT(2) ;
  return res ;
}
```

Comparing the R API to Rcpp: Functions

Using the basic C API for R.

```
#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)
     )
   )
  ));
  GetRNGState();
  SEXP res = PROTECT(eval(call,R_GlobalEnv));
  PutRNGState();
  UNPROTECT(2) ;
  return res ;
}
```

Or using **Rcpp**.

```
#include <Rcpp.h>
extern "C" SEXP callback(){
  RNGScope s;
  Language l = Language("rnorm",
                        3, 10.0, 20.0);
  return l.eval(R_GlobalEnv) ;
}
```

Comparing the R API to Rcpp: Functions

Using the basic C API for R.

```
#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)
     )
    )
  ));
  GetRNGState();
  SEXP res = PROTECT(eval(call,R_GlobalEnv));
  PutRNGState();
  UNPROTECT(2) ;
  return res ;
}
```

Or using **Rcpp**.

```
#include <Rcpp.h>
extern "C" SEXP callback(){
  RNGScope s;
  Language l = Language("rnorm",
                        3, 10.0, 20.0);
  return l.eval(R_GlobalEnv) ;
}
```

or using **Rcpp** differently

```
#include <Rcpp.h>
extern "C" SEXP callback(){
  RNGScope s;
  Function f = Function("rnorm");
  return f(3, 10, 20);
}
```

Comparing the R API to Rcpp: Functions

Using the basic C API for R.

```
#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)
       )
     )
  ));
  GetRNGState();
  SEXP res = PROTECT(eval(call,R_GlobalEnv));
  PutRNGState();
  UNPROTECT(2) ;
  return res ;
}
```

Or using **Rcpp**.

```
#include <Rcpp.h>
extern "C" SEXP callback(){
  RNGScope s;
  Language l = Language("rnorm",
                        3, 10.0, 20.0);
  return l.eval(R_GlobalEnv) ;
}
```

or using **Rcpp** differently

```
#include <Rcpp.h>
extern "C" SEXP callback(){
  RNGScope s;
  Function f = Function("rnorm");
  return f(3, 10, 20);
}
```

or using **Rcpp** sugar

```
#include <Rcpp.h>
extern "C" SEXP callback(){
  RNGScope s;
  return rnorm(3, 10, 20);
}
```

Comparing the R API to Rcpp: Functions

Using the basic C API for R.

```
#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)
     )
    )
  ));
  GetRNGState();
  SEXP res = PROTECT(eval(call,R_GlobalEnv));
  PutRNGState();
  UNPROTECT(2) ;
  return res ;
}
```

Or using **Rcpp**.

```
#include <Rcpp.h>
extern "C" SEXP callback(){
  RNGScope s;
  Language l = Language("rnorm",
                        3, 10.0, 20.0);
  return l.eval(R_GlobalEnv) ;
}
```

or using **Rcpp** differently

```
#include <Rcpp.h>
extern "C" SEXP callback(){
  RNGScope s;
  Function f = Function("rnorm");
  return f(3, 10, 20);
}
```

or using **Rcpp** sugar

```
#include <Rcpp.h>
extern "C" SEXP callback(){
  RNGScope s;
  return rnorm(3, 10, 20);
}
```

or using R:

```
res <- rnorm(3, 10.0, 20.0)
```

Comparing the R API to Rcpp: Lists

Using the basic C API for R.

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

extern "C" SEXP listex(){
  SEXP res = PROTECT(allocVector(VECSXP, 2));
  SEXP x1  = PROTECT(allocVector(REALSXP, 2));
  SEXP x2  = PROTECT(allocVector(INTSXP, 2));
  SEXP klass = PROTECT(mkString("foobar")) ;

  double* px1 = REAL(x1);
  px1[0] = 0.5;
  px1[1] = 1.5;
  int* px2 = INTEGER(x2);
  px2[0] = 2;
  px2[1] = 3;

  SET_VECTOR_ELT(res, 0, x1);
  SET_VECTOR_ELT(res, 1, x2);
  setAttrib(res, install("class"), klass) ;

  UNPROTECT(4) ;
  return res ;
}
```

Comparing the R API to Rcpp: Lists

Using the basic C API for R.

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

extern "C" SEXP listex(){
  SEXP res = PROTECT(allocVector(VECSXP, 2));
  SEXP x1  = PROTECT(allocVector(REALSXP, 2));
  SEXP x2  = PROTECT(allocVector(INTSXP, 2));
  SEXP klass = PROTECT(mkString("foobar")) ;

  double* px1 = REAL(x1);
  px1[0] = 0.5;
  px1[1] = 1.5;
  int* px2 = INTEGER(x2);
  px2[0] = 2;
  px2[1] = 3;

  SET_VECTOR_ELT(res, 0, x1);
  SET_VECTOR_ELT(res, 1, x2);
  setAttrib(res, install("class"), klass) ;

  UNPROTECT(4) ;
  return res ;
}
```

Or using **Rcpp**.

```
#include <Rcpp.h>
extern "C" SEXP listex2(){
  NumericVector x=NumericVector::create(.5,1.5);
  IntegerVector y=IntegerVector::create(2, 3);
  List res      =List::create(x, y);
  res.attr("class") = "foobar";
  return res;
}
```

Comparing the R API to Rcpp: Lists

Using the basic C API for R.

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

extern "C" SEXP listex(){
  SEXP res = PROTECT(allocVector(VECSXP, 2));
  SEXP x1  = PROTECT(allocVector(REALSXP, 2));
  SEXP x2  = PROTECT(allocVector(INTSXP, 2));
  SEXP klass = PROTECT(mkString("foobar")) ;

  double* px1 = REAL(x1);
  px1[0] = 0.5;
  px1[1] = 1.5;
  int* px2 = INTEGER(x2);
  px2[0] = 2;
  px2[1] = 3;

  SET_VECTOR_ELT(res, 0, x1);
  SET_VECTOR_ELT(res, 1, x2);
  setAttrib(res, install("class"), klass) ;

  UNPROTECT(4) ;
  return res ;
}
```

Or using **Rcpp**.

```
#include <Rcpp.h>
extern "C" SEXP listex2(){
  NumericVector x=NumericVector::create(.5,1.5);
  IntegerVector y=IntegerVector::create(2, 3);
  List res       =List::create(x, y);
  res.attr("class") = "foobar";
  return res;
}
```

Or using R:

```
ex4 <- function() {
  x <- c(0.5, 1.5)
  y <- c(2L, 3L)
  r <- list(x, y)
  class(r) <- "foobar"
  r
}
```

Outline

1. Why would we extend R with C++?
2. How can Rcpp help us?
3. What can we do with Rcpp?
4. What else should we know about Rcpp?
5. Who is using Rcpp?
6. And One More Thing

So what do we do?

Recall that we said the *why* boiled down to speed (which we will focus on), new things and object references.

We will look at a few examples which (re-)introduce **Rcpp** concepts and extensions, and demonstrate the gains that can be had:

- Recursive functions
- Simple looped data generation
- A simple MCMC example
- The OLS horse race

Rcpp essentials in one page

The earlier examples showed that **Rcpp**

- can both receive entire R objects: vectors, matrices, list, ... as well as basic C++ types int, double, string, ...
- can create and return R objects easily: vectors, list, functions, matrices, ...
- this makes interfacing C++ code from R so much easier
- the **inline** package facilitates prototyping

What we haven't shown (but is extensively documented):

- how to extend **Rcpp** to wrap around other class libraries: **RcppArmadillo**, **RcppEigen**, **RcppGSL**, ...
- how to use **Rcpp** in your own packages.

Computing the Fibonacci sequence faster

A question on theStackOverflow site lead to short blog post, and an example now included with **Rcpp**. The R function

```
fibR <- function(x) {
    if (x == 0) return(0);
    if (x == 1) return(1);
    return (fibR(x - 1) + fibR(x - 2));
}
```

can be replaced with this **Rcpp/inline** construct:

```
incltxt <- '
int fibonacci(const int x) {
   if (x == 0) return(0);
   if (x == 1) return(1);
   return (fibonacci(x - 1)) + fibonacci(x - 2);
}'
fibRcpp <- cxxfunction(signature(xs="int"),
                       plugin="Rcpp",
                       incl=incltxt,
                       body='
   int x = Rcpp::as<int>(xs);
   return Rcpp::wrap( fibonacci(x) );
')
```

Computing the Fibonacci sequence faster: Result

Running the `examples/Misc/fibonacci.r` example in the **Rcpp** package:

```
edd@max:~$ r svn/rcpp/pkg/Rcpp/inst/examples/Misc/fibonacci.r
Loading required package: inline
Loading required package: methods
Loading required package: compiler
         test replications elapsed relative user.self sys.self
3 fibRcpp(N)            1   0.095   1.0000      0.09     0.00
1    fibR(N)            1  65.813 692.7684     65.73     0.04
2   fibRC(N)            1  65.928 693.9789     65.89     0.00
edd@max:~$
```

95 milliseconds for **Rcpp**, versus 65.8 and 65.9 seconds for R and byte-compiled R — a 690-fold gain.

(Of course, even better gains come from switching to an iterative algorithm.)

Simulating Vector Auto Regression (VAR): R

Lance Bachmeier shared an example from his graduate econometrics class which we worked in to an example in **RcppArmadillo** as well as a short blog post.

```
## parameter and error terms used throughout
a <- matrix(c(0.5,0.1,0.1,0.5),nrow=2)
e <- matrix(rnorm(10000),ncol=2)

## Let's start with the R version
rSim <- function(coeff, err) {
  simd <- matrix(0, nrow(err), ncol(err))
  for (r in 2:nrow(err)) {
    simd[r,] = coeff %*% simd[r-1,] + err[r,]
  }
  return(simd)
}

rData <- rSim(a, e)      # generated by R
```

Simulating Vector Auto Regression (VAR): C++

```
## Now load 'inline' to compile C++ code on the fly
suppressMessages(require(inline))
code <- '
  arma::mat coeff = Rcpp::as<arma::mat>(a);
  arma::mat errors = Rcpp::as<arma::mat>(e);
  int m = errors.n_rows; int n = errors.n_cols;
  arma::mat simdata(m,n);
  simdata.row(0) = arma::zeros<arma::mat>(1,n);
  for (int row=1; row<m; row++) {
    simdata.row(row) = simdata.row(row-1)*trans(coeff)+errors.row(row);
  }
  return Rcpp::wrap(simdata);
'

## create the compiled function
rcppSim <- cxxfunction(signature(a="numeric",e="numeric"),
                       code,plugin="RcppArmadillo")

rcppData <- rcppSim(a,e)                  # generated by C++ code

stopifnot(all.equal(rData, rcppData))    # checking results
```

Simulating Vector Auto Regression (VAR): Result

We run the example from in the **RcppArmadillo** sources:

```
edd@max:~$ r svn/rcpp/pkg/RcppArmadillo/inst/examples/varSimulation.r
            test replications elapsed  relative user.self sys.self
1  rcppSim(a, e)          100   0.032   1.00000      0.02     0.01
3 compRsim(a, e)          100   2.113  66.03125      2.09     0.01
2     rSim(a, e)          100   4.622 144.43750      4.63     0.00
edd@max:~$
```

Rcpp provides a 140-fold gain over uncompiled R; the byte compiler (new with R 2.13.0) helps by roughly halfing the computation yet is still beat by a factor of 60+ by the C++ code.

MCMC Gibbs Sampler

Sanjog Misra pointed me to an example by Darren Wilkinson (comparing MCMC via a few languages) and a first implementation which we reworked into what beccame another **Rcpp** example (see `GibbsCode`).

Here, the bivariate distribution

$$f(x, y) = k \cdot x^2 \cdot e^{-xy^2 - y^2 + 2y - 4x}$$

is sampled via two conditional distributions:

$$\begin{aligned} f(x|y) &= x^2 e^{-x(4+y^2)} && \text{// Gamma} \\ f(y|x) &= e^{-0.5 \cdot 2(x+1) \cdot (y^2 - 2y/(x+1))} && \text{// Gaussian} \end{aligned}$$

which cannot be vectorised due to interdependence.

MCMC Gibbs Sampler: R Version

The R version is pretty straightforward:

```
## Here is the actual Gibbs Sampler
## This is Darren Wilkinsons R code (with the corrected variance)
## But we are returning only his columns 2 and 3 as the 1:N sequence
## is never used below
Rgibbs <- function(N,thin) {
    mat <- matrix(0,ncol=2,nrow=N)
    x <- 0
    y <- 0
    for (i in 1:N) {
        for (j in 1:thin) {
            x <- rgamma(1,3,y*y+4)
            y <- rnorm(1,1/(x+1),1/sqrt(2*(x+1)))
        }
        mat[i,] <- c(x,y)
    }
    mat
}
```

as is the byte-compiled variant:

```
## We can also try the R compiler on this R function
RCgibbs <- cmpfun(Rgibbs)
```

MCMC Gibbs Sampler: Rcpp Version

```
## Now for the Rcpp version -- Notice how easy it is to code up!
gibbscode <- '
  using namespace Rcpp;    // inline does that for us already
  // n and thin are SEXPs which the Rcpp::as function maps to C++ vars
  int N   = as<int>(n);
  int thn = as<int>(thin);
  int i,j;
  NumericMatrix mat(N, 2);

  RNGScope scope;          // Initialize Random number generator

  // The rest of the code follows the R version
  double x=0, y=0;
  for (i=0; i<N; i++) {
    for (j=0; j<thn; j++) {
      x = ::Rf_rgamma(3.0,1.0/(y*y+4));
      y = ::Rf_rnorm(1.0/(x+1),1.0/sqrt(2*x+2));
    }
    mat(i,0) = x;
    mat(i,1) = y;
  }
  return mat;              // Return to R
'
# Compile and Load
RcppGibbs <- cxxfunction(signature(n="int", thin = "int"),
                         gibbscode, plugin="Rcpp")
```

MCMC Gibbs Sampler: Results

The results are again quite favourable to **Rcpp**, beating even the byte-compiled variant by a factor of 24:

```
R> ## use rbenchmark package
R> N <- 10000
R> thn <- 100
R> res <- benchmark(Rgibbs(N, thn),
+                  RCgibbs(N, thn),
+                  RcppGibbs(N, thn),
+                  columns=c("test", "replications", "elapsed",
+                            "relative", "user.self", "sys.self"),
+                  order="relative",
+                  replications=10)
R> print(res)
                test replications elapsed relative user.self sys.self
3 RcppGibbs(N, thn)           10   2.972   1.0000      2.97        0
2   RCgibbs(N, thn)           10  72.919  24.5353     72.83        0
1    Rgibbs(N, thn)           10 104.830  35.2725    104.72        0
R>
```

NB: Not shown are numbers from a GSL version which is even faster due to a much faster Gamma distribution RNG in the GSL.

Faster linear regressions

This is a recurrent theme for me going back to a question by Ivo Welch many years ago: how does one do `lm()` faster when one also wants standard errors (to simulate test size / power trade-offs) ?

I had written first versions using the first-generation, more basic **Rcpp** against the GSL, then with Armadillo, later Rcpp**Armadillo** and now Eigen.

There is an older example in the **Rcpp** package which predates the add-on packages **RcppGSL** and **RcppArmadillo** – both of which implement faster `fastLm()` functions.

But the state-of-the-art variant is in the vignette of the **RcppEigen** package and part of a paper Doug Bates and I just submitted.

Faster linear regressions: Old Comparison

These implementation predate the RcppArmadillo and RcppGSL packages

Using the ancient Longley dataset:

```
edd@max:~/svn/rcpp/pkg/Rcpp/inst/examples/FastLM$ ./benchmarkLongley.r
For Longley
                 lm        lm.fit          lmGSL        lmArma
results 0.001666667 1.488889e-04 2.555556e-05 5.222222e-05
ratios  1.000000000 1.119403e+01 6.521739e+01 3.191489e+01
         lm    lm.fit     lmGSL    lmArma
results 600 6716.418 39130.43 19148.94
edd@max:~/svn/rcpp/pkg/Rcpp/inst/examples/FastLM$
```

Using simulated data:

```
edd@max:~/svn/rcpp/pkg/Rcpp/inst/examples/FastLM$ ./benchmark.r
For n=25000 and k=9
               lm       lm.fit       lmGSL        lmArma
results 0.1669111  0.01412222 0.03103333  0.009722222
ratios  1.0000000 11.81904013 5.37844612 17.168000000
              lm    lm.fit     lmGSL    lmArma
results 5.991213 70.81039 32.22342 102.8571
edd@max:~/svn/rcpp/pkg/Rcpp/inst/examples/FastLM$
```

Faster linear regressions: Recent Comparison

Bates, Eddelbuettel (2011), "Fast + Elegant Numerical Lin. Algebra Using RcppEigen"

Method	Relative	Elapsed	User	Sys
LDLt	1.00	1.18	1.17	0.00
LLt	1.01	1.19	1.17	0.00
SymmEig	2.76	3.25	2.70	0.52
QR	6.35	7.47	6.93	0.53
arma	6.60	7.76	25.69	4.47
PivQR	7.15	8.41	7.78	0.62
lm.fit	11.68	13.74	21.56	16.79
GESDD	12.58	14.79	44.01	10.96
SVD	44.48	52.30	51.38	0.80
GSL	150.46	176.95	210.52	149.86

Table: `lmBenchmark` (from the **RcppEigen** package) results on a desktop computer for the default size, $100,000 \times 40$, full-rank model matrix running 20 repetitions for each method. Times (Elapsed, User and Sys) are in seconds.

Outline

Rcpp Sugar: vectorised C++ expressions

Rcpp sugar brings *syntactic sugar* to C++ / Rcpp programming:

- vectorized expression similar to R: `ifelse(...)`
- all the standard binary and arithmetic operators
- functions such as `any()`, `all()`, `seq_along()`, `pmin()`, `pmax()`, ... and even `sapply()` and `lapply()`
- mathematic functions: `abs()`, `exp()`, `log()`, ...
- statistical d/q/p/r functions on beta, binom, cauchy, chisq, exp, f, gamma, ... distributions

Details are in the twelve-page vignette “Rcpp-sugar”.

Rcpp Modules: Just declaring interfaces

Rcpp Modules are inspired by a Boost.Python C++ library. Some of their key features allow us

- expose functions just by declaring the interface
- expose classes similarly just by declarations
- this includes support for constructors, private and public fields, read-only as well as read-write access and more.

The “Rcpp-modules” vignette has details, and shows how to deploy Modules in your own package.

Packages: How to deploy Rcpp beyond inline

Rcpp provides a function `Rcpp.package.skeleton()` which extends the base R functions after which it is modeled. It creates

- basic package directory structure
- necessary files such as `src/Makevars` and `src/Makevars.win`, `NAMESPACE` and more
- a set C++ function files (header and sources), and an R function to call it
- simple documentation files

The vignette “Rcpp-package” discusses this in more detail.

Outline

CRAN Packages using Rcpp

As of early December 2011, the following packages on CRAN use Rcpp and hence depend on it:

> ***auteur***, ***bcp***, ***bfa***, ***bifactorial***, ***cda***, ***fastGHQuad***, ***fdaMixed***, ***GUTS***, ***highlight***, ***KernSmoothIRT***, ***LaF***, ***maxent***, ***minqa***, ***multmod***, ***mvabund***, ***NetworkAnalysis***, ***nfda***, ***orQA***, ***parser***, ***phylobase***, ***planar***, ***Rclusterpp***, ***RcppArmadillo***, ***RcppBDT***, ***RcppClassic***, ***RcppDE***, ***RcppEigen***, ***RcppExamples***, ***RcppGSL***, ***rgam***, ***RInside***, ***rococo***, ***RProtoBuf***, ***RQuantLib***, ***RSNNS***, ***RSofia***, ***rugarch***, ***RVowpalWabbit***, ***SBSA***, ***sdcTable***, ***simFrame***, ***spacodiR***, ***termstrc***, ***unmarked***, ***VIM***, ***wordcloud***

CRAN Packages using Rcpp

We can identify some broad categories among these packages:

- packages which re-implement already existing R code in C++ for greater speed: **termstr**, **wordcloud**
- packages which connect to external libraries: **RQuantLib**, **RProtoBuf**, **RSNNS**, **RSofia**, **RVowpalWabbit**
- packages directly related to Rcpp providing glue to other libraries: **RcppArmadillo**, **RcppEigen**, **RcppGSL**
- packages using Rcpp Modules to easily interface C++ code: **RcppBDT**, **cds**, **planar**

Outline

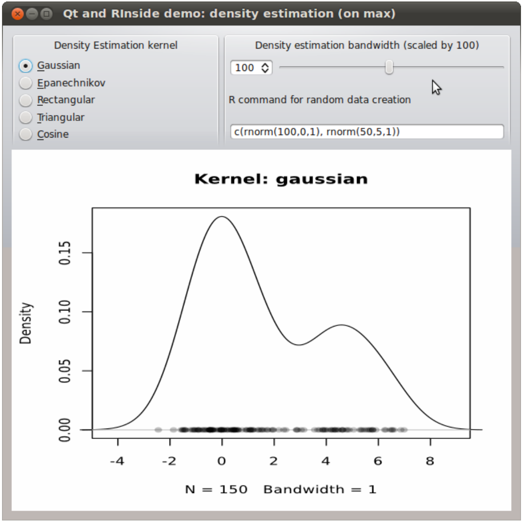
RInside makes it trivial to embed R

This is rinside_sample11.cpp from the RInside examples

```
// Simple example motivated by post from Paul Smith <phhs80@gmail.com>
// to r-help on 06 Mar 2011
//
// Copyright (C) 2011   Dirk Eddelbuettel and Romain Francois

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

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

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

  // evaluate an R expression with curve()
  // because RInside defaults to interactive=false we use a file
  std::string cmd = "tmpf <- tempfile('curve'); "
    "png(tmpf); curve(x^2, -10, 10, 200); dev.off();"
    "tmpf";
  // by running parseEval, we get the last assignment back, here the filename
  std::string tmpfile = R.parseEval(cmd);

  std::cout << "Could now use plot in " << tmpfile << std::endl;
  unlink(tmpfile.c_str());   // cleaning up

  // alternatively, by forcing a display we can plot to screen
  cmd = "x11(); curve(x^2, -10, 10, 200); Sys.sleep(30);";
  R.parseEvalQ(cmd);         // parseEvalQ evluates without assignment

  exit(0);
}
```

RInside allows us to embed R in desktop applications

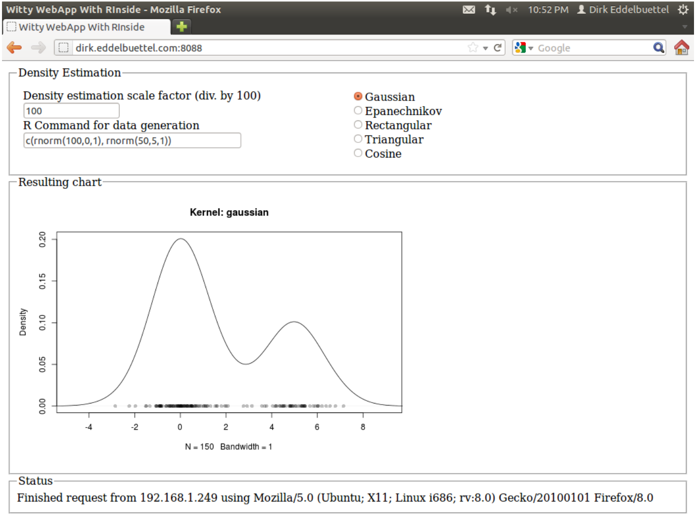
This uses the Qt C++ toolkit (cf examples/qt in RInside)



This example is discussed more fully on my blog, and the full sources are included in the RInside package.

RInside also allows us to embed R in web applications

This uses the Wt C++ toolkit (cf examples/wt in RInside)



Also on blog, sources in SVN and in next RInside release.

That's it for today

For more information:

- the eight pdf vignettes in the **Rcpp** package (which includes our *Journal of Statistical Software* paper)
- Dirk's site, code section and blog: `http://dirk.eddelbuettel.com`
- Romain site: `http://romainfrancois.blog.free.fr/index.php?category/R-package/Rcpp`
- CRAN page(s): `http://cran.r-project.org/web/packages/Rcpp/index.html`
- The rcpp-devel mailing list.