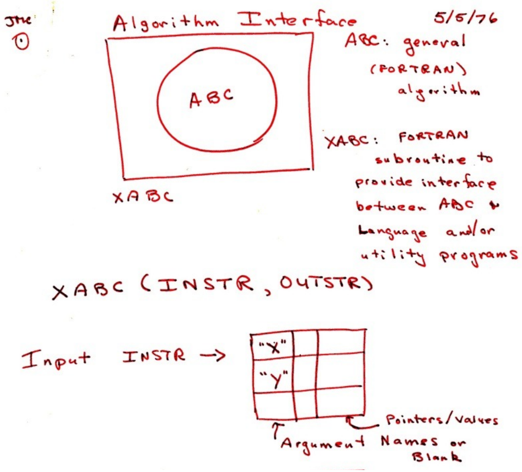
A Brief Introduction to Rcpp

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

Chicago R Users Group
Chicago, IL
8 August 2013

Outline

A "vision" from Bell Labs from 1976



Source: John Chambers' talk at Stanford in October 2010; personal correspondence.

Outline

Uses only standard R tools to build packages

Depending on the platform, one needs

Windows the Rtools kit for Windows, properly installed – see CRAN, the Installation manual and many tutorials; the **installr** package may help

OS X the Xcode *command-line tools* (plus possibly the Fortran compiler) – see Simon's pages

Linux generally just work out of the box

Several environments can be used to work with Rcpp – RStudio is very popular.

No additional requirements for Rcpp beyond *being able to compile R packages*.

Easy to test:

```
library(Rcpp)
## evaluate a C++ expression, retrieve result
evalCpp("2 + 2")

## [1] 4

## a little fancier
evalCpp("std::numeric_limits<double>::max()")

## [1] 1.798e+308

## create ad-hoc R function 'square'
cppFunction('int square(int x) { return x*x;}')
square(7L)

## [1] 49
```

What are some of the key features of Rcpp?

Easy to use it really does not have to be that complicated – we will look at a few examples

Expressive it allows you to write *vectorised* C++ using *Rcpp Sugar*

Seamless it gives access to all R objects: vector, matrix, list, S3/S4/RefClass, Environment, Function, ...

Speed gains for a variety of tasks Rcpp can excel precisely where R struggles: loops, function calls, ...

Extensions greatly facilitates access to external libraries using eg *Rcpp modules* (but we will not have time for a walkthrough)

An Introductory Example

Consider a function defined as

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

An Introductory Example: Simple R Implementation

R implementation and use:

```
f <- function(n) {
    if (n < 2) return(n)
    return(f(n-1) + f(n-2))
}
## Using it on first 11 arguments
sapply(0:10, f)

##  [1]  0  1  1  2  3  5  8 13 21 34 55
```

An Introductory Example: Timing R Implementation

Timing:

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

##     test replications elapsed relative
## 1 f(10)          100   0.030     1.00
## 2 f(15)          100   0.332    11.07
## 3 f(20)          100   3.679   122.63
```

An Introductory Example: C++ Implementation

```
int g(int n) {
    if (n < 2) return(n);
    return(g(n-1) + g(n-2));
}
```

Deployed as:

```
library(Rcpp)
cppFunction("
  int g(int n) {
    if (n < 2) return(n);
    return(g(n-1) + g(n-2));
}")
## Using it on first 11 arguments
sapply(0:10, g)
```

An Introductory Example: Comparing timing

Timing:

```
library(rbenchmark)
benchmark(f(20), g(20))[,1:4]

##    test replications elapsed relative
## 1 f(20)          100   3.769    538.4
## 2 g(20)          100   0.007      1.0
```

Usually around a nice 600-fold gain.

Type mapping

Standard R types (integer, numeric, list, function, ... and compound objects) are mapped to corresponding C++ types using extensive template meta-programming – it just works:

```
library(Rcpp)
cppFunction("
  NumericVector logabs(NumericVector x) {
    return log(abs(x));
  }
")
logabs(seq(-5, 5, by=2))

## [1] 1.609 1.099 0.000 0.000 1.099 1.609
```

Also note: vectorized C++!

Type mapping also with C++ STL types

Use of `std::vector<double>` and STL algorithms:

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

inline double f(double x) { return ::log(::fabs(x)); }

// [[Rcpp::export]]
std::vector<double> logabs2(std::vector<double> x) {
  std::transform(x.begin(), x.end(), x.begin(),  f);
  return x;
}
```

Type mapping also with C++ STL types

Used via

```
library(Rcpp)
sourceCpp("code/logabs2.cpp")
logabs2(seq(-5, 5, by=2))

## [1] 1.609 1.099 0.000 0.000 1.099 1.609
```

Type mapping is seamless

Simple outer product of a column vector (using RcppArmadillo):

```
cppFunction("arma::mat v(arma::vec a) {
                return a*a.t();
            }", depends="RcppArmadillo")
v(1:4)

##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    2    4    6    8
## [3,]    3    6    9   12
## [4,]    4    8   12   16
```

This uses implicit conversion via `as<>` and `wrap` – cf package vignette Rcpp-extending.

Well-know packages using Rcpp

Amelia by Gary King et al: Multiple Imputation from cross-section, time-series or both; uses Rcpp and RcppArmadillo

forecast by Rob Hyndman et al: Time-series forecasting including state space and automated ARIMA modeling; uses Rcpp and RcppArmadillo

RStan by Andrew Gelman et al: Rcpp helps with automatic model parsing / generation for MCMC / Bayesian modeling

rugarch by Alexios Ghalanos: Sophisticated financial time series models using Rcpp and RcppArmadillo

bigviz by Hadley Wickham: High-performance visualization of datasets in the 10-100 million observations range

Outline

Basic Usage: evalCpp

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

This allows us to quickly check C++ constructs.

```
evalCpp( "2 * M_PI" )

## [1] 6.283
```

Basic Usage: `cppFunction()`

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

```
cppFunction("
    int useCpp11() {
        auto x = 10;
        return x;
}", plugins=c("cpp11"))
useCpp11()  # same identifier as C++ function

## [1] 10
```

Basic Usage: `sourceCpp()`

`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).

We are also starting to provide other compiler features via plugins. A first plugin to enable C++11 support was added in Rcpp 0.10.3.

Basic Usage: Packages

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

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

The vignette Rcpp-package has fuller details.

As of August 2013, there are 130 packages on CRAN which use Rcpp, and a further 13 on BioConductor — all with working, tested, and reviewed examples.

Outline

Syntactive 'sugar': Simulating π in R

Basic idea: for point (x, y), compute distance to origin. Do so repeatedly, and the ratio of points below one to number N of simulations will approach $\pi/4$ as we fill the area of one quarter of the unit circle.

```
piR <- function(N) {
    x <- runif(N)
    y <- runif(N)
    d <- sqrt(x^2 + y^2)
    return(4 * sum(d <= 1.0) / N)
}

set.seed(5)
sapply(10^(3:6), piR)

## [1] 3.156 3.155 3.139 3.141
```

Syntactive 'sugar': Simulating π in C++

The neat thing about *Rcpp sugar* is that it enables us to write C++ code that looks almost as compact.

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

// [[Rcpp::export]]
double piSugar(const int N) {
    RNGScope scope;  // ensure RNG gets set/reset
    NumericVector x = runif(N);
    NumericVector y = runif(N);
    NumericVector d = sqrt(x*x + y*y);
    return 4.0 * sum(d <= 1.0) / N;
}
```

Apart from RNG set/reset, the code is essentially identical.

Syntactive 'sugar': Simulating π

And by using the same RNG, so are the results.

```
sourceCpp("code/piSugar.cpp")
set.seed(42); a <- piR(1.0e7)
set.seed(42); b <- piSugar(1.0e7)
identical(a,b)

## [1] TRUE

print(c(a,b), digits=7)

## [1] 3.140899 3.140899
```

Syntactive 'sugar': Simulating π

Here, the performance gain is less dramatic as the R code is already vectorised:

```
library(rbenchmark)
benchmark(piR(1.0e6), piSugar(1.0e6))[,1:4]

##              test replications elapsed relative
## 1      piR(1e+06)          100  11.731    2.184
## 2 piSugar(1e+06)          100   5.372    1.000
```

More about Sugar is in the package vignette Rcpp-sugar.

Outline

MCMC and Gibbs Samplers

Markov chain Monte Carlo, and the Gibbs sampler in particular are popular to simulate posterior densities.

A set of posts by Darren Wilkinson spawned a cottage industry of comparisons among languages, dialects, variants, ...

We will briefly revisit it here too.

Gibbs Sampler Setup

The example by Darren shows a simple MCMC Gibbs sampler of this bivariate density::

$$f(x,y) = kx^2 \exp(-xy^2 - y^2 + 2y - 4x)$$

with conditional distributions

$$\begin{aligned} f(x|y) &\sim \text{Gamma}(3, y^2 + 4) \\ f(y|x) &\sim N\left(\frac{1}{1+x}, \frac{1}{2(1+x)}\right) \end{aligned}$$

i.e. we need repeated RNG draws from both a Gamma and a Gaussian distribution.

Gibbs Sampler: R Code

```
## The actual Gibbs Sampler
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
}
library(compiler)  ## to byte-compile
RCgibbs <- cmpfun(Rgibbs)
```

Gibbs Sampler: C++ Code

```
#include <Rcpp.h>  // load Rcpp
using namespace Rcpp;  // shorthand
// [[Rcpp::export]]
NumericMatrix RcppGibbs(int n, int thn) {
    int i,j;
    NumericMatrix mat(n, 2);
    // 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 = R::rgamma(3.0,1.0/(y*y+4));
            y = R::rnorm(1.0/(x+1),1.0/sqrt(2*x+2));
        }
        mat(i,0) = x;
        mat(i,1) = y;
    }
    return mat;              // Return to R
}
```

Gibbs Sampler: Benchmark

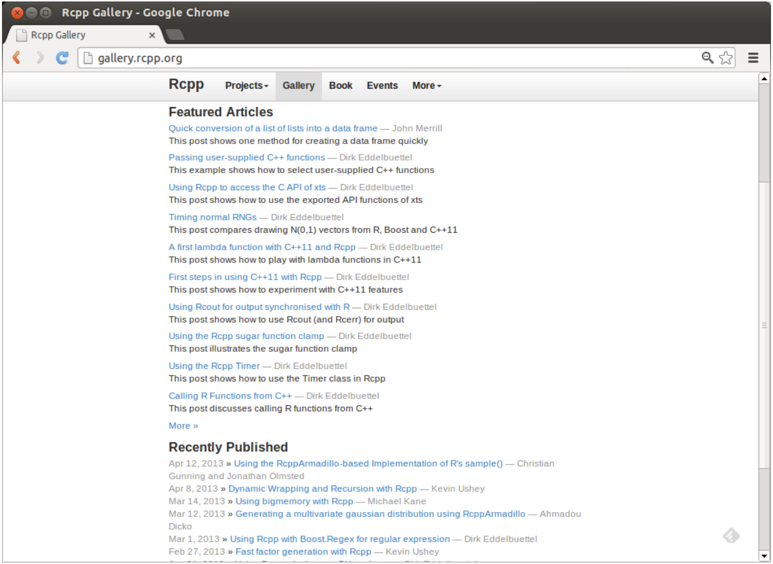
```
source("code/gibbs.R")
sourceCpp("code/gibbs.cpp")
library(rbenchmark)
benchmark(Rgibbs(1000,100),
          RCgibbs(1000,100),
          RcppGibbs(1000,100),
          replications=10,
          order="relative")[,c(1,3:4)]

##                   test elapsed relative
## 3 RcppGibbs(1000, 100)   0.288     1.00
## 2   RCgibbs(1000, 100)  10.769    37.39
## 1    Rgibbs(1000, 100)  14.320    49.72
```

Outline

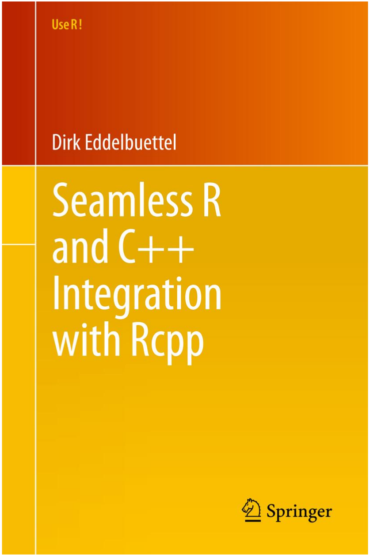
Documentation

- The package comes with **eight pdf vignettes**, and numerous help pages.
- The introductory vignettes are now **published** (Rcpp and RcppEigen in *J Stat Software*, RcppArmadillo in *Comp. Stat.& Data Anal.*).
- The **rcpp-devel** list is *the* recommended resource, generally very helpful, and fairly low volume.
- By now **StackOverflow** has a fair number of posts too.
- And a number of blog posts introduce/discuss features.

Rcpp Gallery



The Rcpp book



In print since June 2013