R and C++ Integration

Abdou Seck

11/3/2017

The R API

The R programming language offers an API to programmers through a set of header files that ship with every R installation.

Essentially everything inside R is represented as a `SEXP` object or `S-expression` in C++. By permitting exchange of such objects between it and C++, R provides programmers with the ability to operate directly on R objects.

There are two main functions that facilitate communication between C/C++ and R: `.C` and `.Call`. The former first appeared in an earlier version of the R language and only supports pointers to basic C types. The latter provides a richer interface and can operate directly on `SEXP` objects.

The following are the main variants of `SEXP` objects:

`REALSXP`: numeric vector

`INTSXP`: integer vector

`LGLSXP`: logical vector

`STRSXP`: character vector

`VECSXP`: list

`CLOSXP`: function (closure)

`ENVSXP`: environment

Compiling and Linking

You can manually compile your C++ code and link the resulting output to your session. But it's easier to use the multitude of R packages that make this process overly simple. Such packages include `inline` and `Rcpp`. Both `inline` and `Rcpp` make it extremely easy to compile and link C code to your current R session. We will be using `Rcpp`.

Example

The following is a C++ function that can be called directly in R after compilation and linking.

```
// In C/C++ ------------------------------------------
#include <R.h>
#include <Rinternals.h>

SEXP add(SEXP a, SEXP b) {
  SEXP result = PROTECT(allocVector(REALSXP, 1));
  REAL(result)[0] = asReal(a) + asReal(b);
  UNPROTECT(1);

  return result;
}
```

The R code calling our function

```
# In R ----------------------------------------
add <- function(a, b) {
  .Call("add", a, b)
}
```

Background

I mainly work with R when it comes to modeling, but any modeling task is made possible by the heavy lifting of data wrangling. Often though, wrangling can be too much of a task for R to deal with; especially when there needs to be some scope conscious looping. By this, I mean any loop wherein we have to worry about previous values. As a result, such operations just cannot be easily ***vectorized***. This is a situation when I feel that I need a *lower* level and much more perfomant language to do such tasks.

Sourcing the files

Calling `source` allows R to run the R code inside the provided file and save the result in the current namespace. `Rcpp::sourceCpp` makes it easy to compile C++ code and link the exported functions to the current R session.

```
source('main_script.R')
```

The script `main_script.R` above contains the following code:

```
lapply(c("Rcpp", "RcppArmadillo", "microbenchmark", "inline", ), function(pkg) {
  if (!require(pkg, character.only = T)) {
    install.packages(pkg)
  } else {
    sprintf("Package %s is already installed and loaded.", pkg)
  }
})

setwd('~/Documents/Personal/Data_Analysis/R/RCpp_Integration/')
# Get the R functions
source('recursive_fib.R')
source('cached_fib.R')
# Get the cpp functions
sourceCpp('recursive_fib.cpp')
sourceCpp('cached_fib.cpp')
```

Naive recursive implementation of Fibonacci series

R

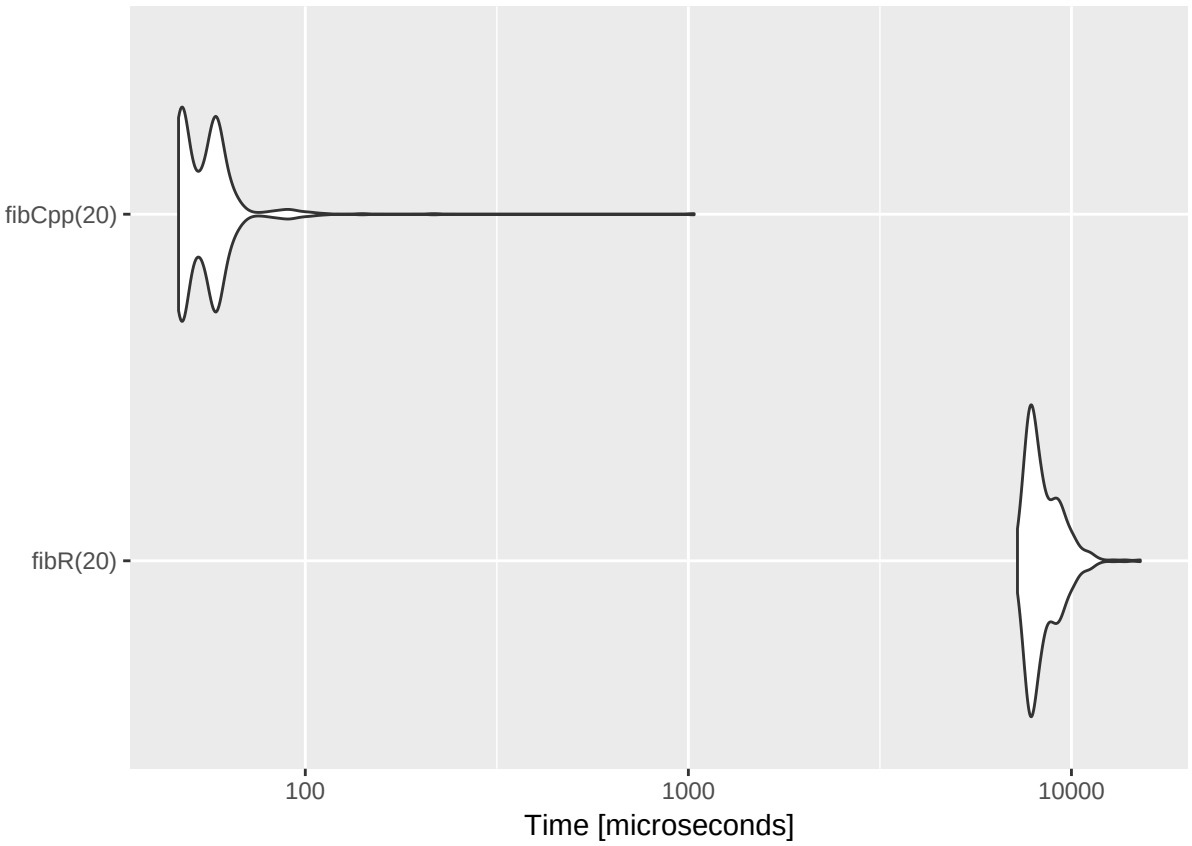
```
fibR <- function(n) {
  if (n == 0) return(0)
  if (n == 1) return(1)
  return(fibR(n-1) + fibR(n-2))
}
```

C++

```
// [[Rcpp::export]]
int fibCpp(const int n) {
  if (n == 0) return 0;
  if (n == 1) return 1;
  return fibCpp(n - 1) + fibCpp(n - 2);
}
```

Benchmarking

```
## Unit: microseconds
##        expr      min        lq       mean   median        uq       max
##    fibR(20) 7229.109 7786.0680 8481.31845 8153.650 9032.1840 15127.795
##  fibCpp(20)   46.681   47.6775   57.90319   55.309   58.9835  1035.562
##  neval
##    500
##    500
```


fibCpp(20)
fibR(20)
100
1000
10000
Time [microseconds]

Cached recursive implementation of Fibonacci series

R

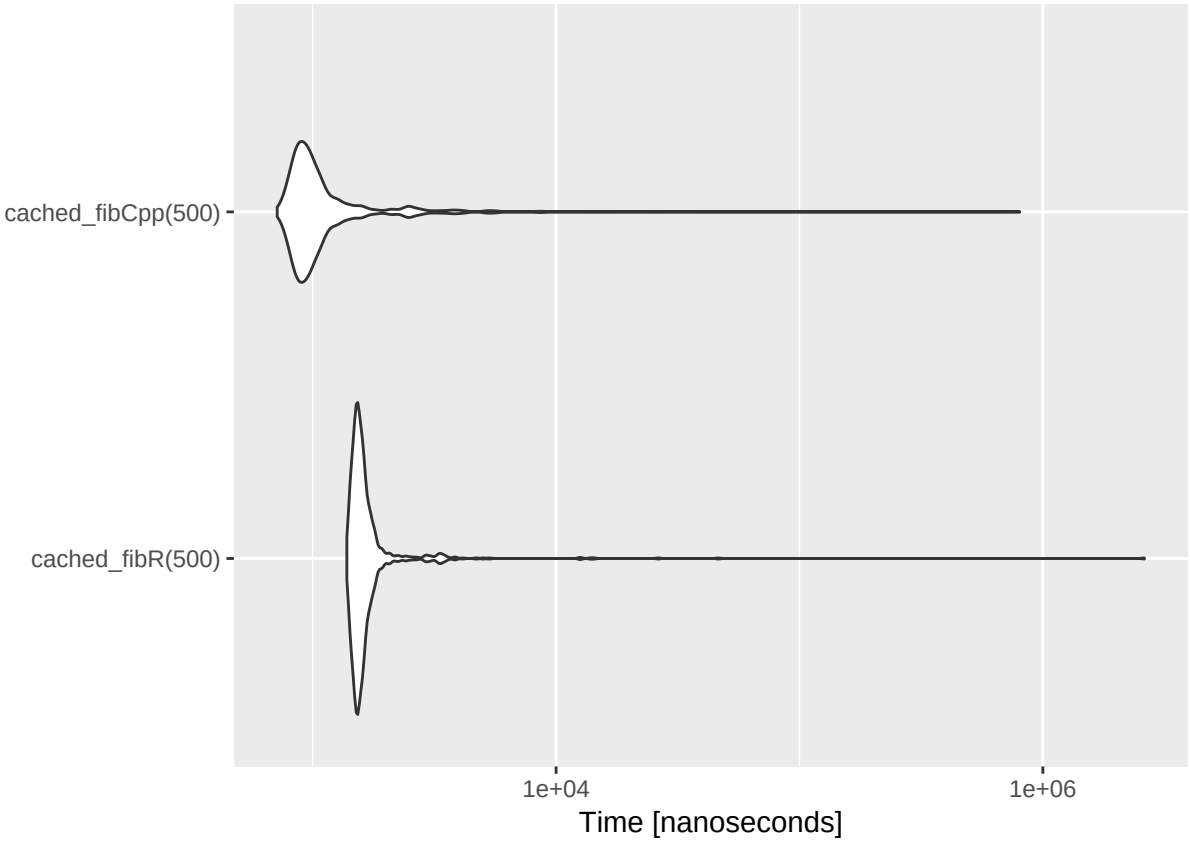
```r
cached_fibR <- local({
  cache <- c(1, 1, rep(NA, 1000))
  f <- function(n) {
    if (n < 0) return(NA)
    if (n > length(cache)) stop("Cannot process for n greater than 1000.")
    if (n == 0) return(0)
    if (n == 1) return(1)
    if (!is.na(cache[n])) return(cache[n])
    ans <- f(n - 1) + f(n - 2)
    cache[n] <<- ans
    ans
  }
})
```

C++

```cpp
#include <Rcpp.h>
#include <algorithm>
#include <vector>
#include <stdexcept>
#include <cmath>
#include <iostream>
using namespace Rcpp;

class Fib {
  public:
    Fib(unsigned int n = 1000) {
      cache.resize(n);
      std::fill(cache.begin(), cache.end(), NAN);
      cache[0] = 0.0;
      cache[1] = 1.0;
    }
    double cached_fibCpp(int x) {
      if (x < 0) {
        return (double) NAN;
      }
      if (x >= (int) cache.size()) {
        throw std::range_error("x too large for implementation");
      }
      if (x < 2) {
        return x;
      }
      if (! ::isnan(cache[x])) return cache[x];
      cache[x] = cached_fibCpp(x - 1) + cached_fibCpp(x - 2);
      return cache[x];

    }
  private:
    std::vector<double> cache;
};

Fib f = Fib(2000);

// [[Rcpp::export]]
double cached_fibCpp(const int a) {
  return f.cached_fibCpp(a);
}
```

Benchmarking

```
## Unit: microseconds
##                 expr   min     lq     mean median     uq      max neval
##    cached_fibR(500) 1.382 1.4970 4.389010  1.558 1.6610 2613.308  1000
##  cached_fibCpp(500) 0.715 0.8725 2.056097  0.960 1.1115  803.504  1000
```



Iterative implementation of Fibonacci series

R

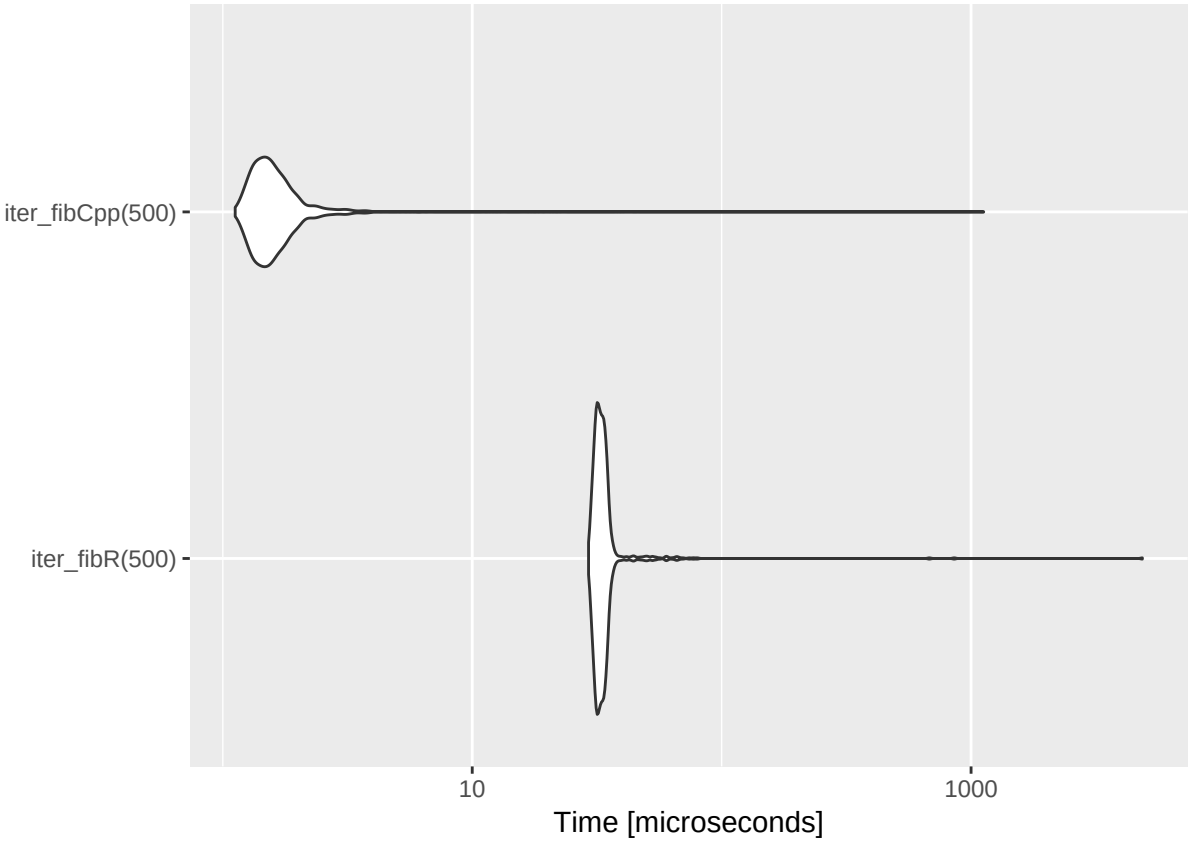
```r
iter_fibR <- function(x) {
  if (x < 0) return(NA)
  if (x < 2) return(x)
  first <- 0
  second <- 1
  third <- 0
  for (i in seq_len(x)) {
    third <- first + second
    first <- second
    second <- third
  }
  first
}
```

C++

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

// [[Rcpp::export]]
double iter_fibCpp(const int x) {
  if (x < 0) return NAN;
  if (x < 2) return x;
  double first = 0.0;
  double second = 1.0;
  double third = 0.0;
  for (int i = 0; i < x; i++) {
    third = first + second;
    first = second;
    second = third;
  }
  return first;
}
```

Benchmarking

```
## Unit: microseconds
##              expr    min      lq      mean  median     uq      max neval
##    iter_fibR(500) 29.268 31.4385 39.808805 32.6430 34.058 4862.766  1000
##  iter_fibCpp(500)  1.121  1.3725  2.844399  1.5305  1.755 1123.231  1000
```



Vector Autogression Model

A VAR is used to capture linear independencies among multiple time series. In sum, it describes the evolution of a set of endogenous variables over the same sample period ($t = 1, ..., T$) as a linear function of only their past values.

Considering the simplest case of a two-dimensional VAR of order one, we can use the following notation:

$$Ax_{t-1} + u_t$$

At time t , the model is comprised of two endogenous variables $x_t = (x_{1t}, x_{2t})$ which are a function of their previous values at $t - 1$ via the coefficient matrix A. When studying the properties of VAR systems, simulation is a tool that is frequently used to assess these models. And, for the simulations, we need to generate suitable data. Due to the interdependence between the two coefficients in the equation above, we can see that vectorization is not going to be possible. Therefore, we will need to loop explicitly.

R

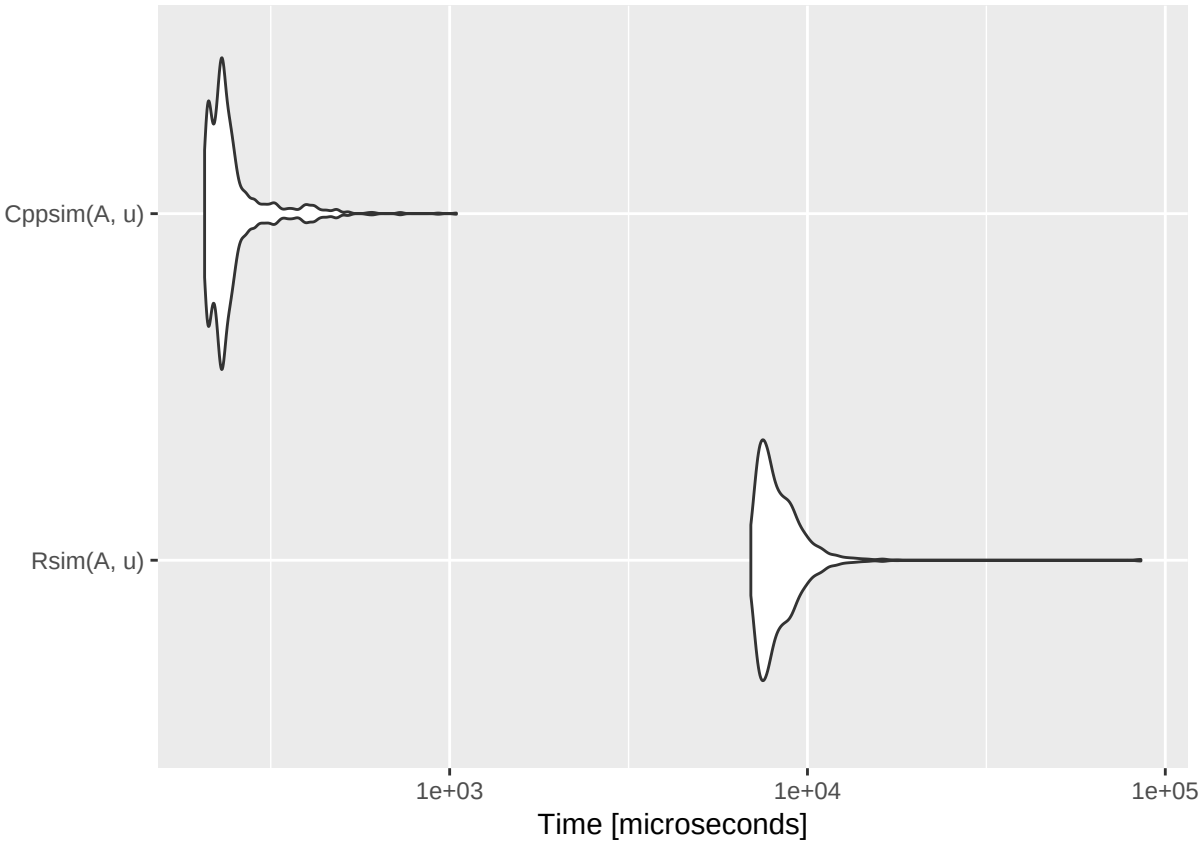
```
A <- matrix(c(0.5, 0.1, 0.1, 0.5), nrow=2)
u <- matrix(rnorm(10000), ncol=2)

Rsim <- function(coeff, errors) {
  simdata <- matrix(0, nrow(errors), ncol(errors))
  for (row in 2:nrow(errors)) {
    simdata[row, ] = coeff %*% simdata[(row-1),] + errors[row, ]
  }
  simdata
}
```

C++

```
library(Rcpp)
library(RcppArmadillo)
suppressMessages(library(inline))

code <- '
  arma::mat coeff = Rcpp::as<arma::mat>(a);
  arma::mat errors = Rcpp::as<arma::mat>(u);
  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);
'
Cppsim <- cxxfunction(signature(a="numeric", u="numeric"), code, plugin="RcppArmadillo");
```

Benchmarking

```
## Unit: microseconds
##          expr      min        lq      mean    median        uq       max
##    Rsim(A, u) 6945.951 7488.2425 8542.6906 7968.2435 8920.4275 85507.243
##  Cppsim(A, u)  206.612  222.2035  252.8797  232.3385  249.8865  1044.693
##  neval
##   1000
##   1000
```



Summary

1. Compile and link using either `inline` or `Rcpp` or both. You may use `R CMD SHLIB` to create a shared object, but you would need to load the shared object into `R` yourself.
2. Don't try to use `C++` for things that are easily vectorized in R.

Thank You.