Higher-Performance R Programming with C++ Extensions

Part 4: Rcpp Gallery Examples

Dirk Eddelbuettel

June 28 and 29, 2017

University of Zürich & ETH Zürich

Rcpp Gallery

Overview

- The *Rcpp Gallery* at `http://gallery.rcpp.org` provides over one-hundred ready-to-run and documented examples.
- It is built on a blog-alike backend in a repository hosted at GitHub.
- You can clone the repository, or just download examples one-by-one.

Simple Examples

Cumulative Sum: vector-cumulative-sum/

A basic looped version:

```
#include <Rcpp.h>
#include <numeric>      // for std::partial_sum
using namespace Rcpp;
// [[Rcpp::export]]
NumericVector cumsum1(NumericVector x) {
    double acc = 0;      // init an accumulator var
    NumericVector res(x.size());  // init result vector
    for (int i = 0; i < x.size(); i++){
         acc += x[i];
         res[i] = acc;
    }
    return res;
}
```

CUMULATIVE SUM: vector-cumulative-sum/

An STL variant:

```
#include <Rcpp.h>
#include <numeric>      // for std::partial_sum
using namespace Rcpp;

// [[Rcpp::export]]
NumericVector cumsum2(NumericVector x) {
    // initialize the result vector
    NumericVector res(x.size());
    std::partial_sum(x.begin(), x.end(),
                     res.begin());
    return res;
}
```

CUMULATIVE SUM: vector-cumulative-sum/

Sugar:

```
// [[Rcpp::export]]
NumericVector cumsum3(NumericVector x) {
    return cumsum(x);  // compute + return result
}
```

Calling an R Function: `r-function-from-c++/`

```cpp
#include <Rcpp.h>

using namespace Rcpp;

// [[Rcpp::export]]
NumericVector callFunction(NumericVector x,
                           Function f) {
    NumericVector res = f(x);
    return res;
}
```

Vector Subsetting: `subsetting/`

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

// [[Rcpp::export]]
NumericVector positives(NumericVector x) {
    return x[x > 0];
}

// [[Rcpp::export]]
List first_three(List x) {
    IntegerVector idx = IntegerVector::create(0, 1, 2);
    return x[idx];
}

// [[Rcpp::export]]
List with_names(List x, CharacterVector y) {
    return x[y];
}
```

VECTOR SUBSETTING: armadillo-subsetting/

```
#include <RcppArmadillo.h>
// [[Rcpp::depends(RcppArmadillo)]]

// [[Rcpp::export]]
arma::mat matrixSubset(arma::mat M) {
    // logical conditionL where is transpose larger?
    arma::umat a = trans(M) > M;
    arma::mat  N = arma::conv_to<arma::mat>::from(a);
    return N;
}

// [[Rcpp::export]]
arma::vec matrixSubset2(arma::mat M) {
    arma::mat Z = M * M.t();
    arma::vec v = Z.elem( arma::find( Z >= 100 ) );
    return v;
}
```

Boost

Boost via BH: a-first-boost-example/

```cpp
// [[Rcpp::depends(BH)]]
#include <Rcpp.h>
#include <boost/math/common_factor.hpp>

// [[Rcpp::export]]
int computeGCD(int a, int b) {
    return boost::math::gcd(a, b);
}

// [[Rcpp::export]]
int computeLCM(int a, int b) {
    return boost::math::lcm(a, b);
}
```

Boost via BH: a-second-boost-example/

```
// [[Rcpp::depends(BH)]]
#include <Rcpp.h>
#include <boost/lexical_cast.hpp>
using boost::lexical_cast;
using boost::bad_lexical_cast;

// [[Rcpp::export]]
std::vector<double> lexicalCast(std::vector<std::string> v) {
    std::vector<double> res(v.size());
    for (int i=0; i<v.size(); i++) {
        try {
            res[i] = lexical_cast<double>(v[i]);
        } catch(bad_lexical_cast &) {
            res[i] = NA_REAL;
        }
    }
    return res;
}
// R> lexicalCast(c("1.23", ".4", "1000", "foo", "42", "pi/4")(
// [1]    1.23    0.40 1000.00       NA   42.00       NA
```

Boost via BH: using-boost-with-bh/

```cpp
// [[Rcpp::depends(BH)]]
#include <Rcpp.h>

// One include file from Boost
#include <boost/date_time/gregorian/gregorian_types.hpp>

using namespace boost::gregorian;

// [[Rcpp::export]]
Rcpp::Date getIMMDate(int mon, int year) {
    // compute third Wednesday of given month / year
    date d = nth_day_of_the_week_in_month(
                       nth_day_of_the_week_in_month::third,
                       Wednesday, mon).get_date(year);
    date::ymd_type ymd = d.year_month_day();
    return Rcpp::Date(ymd.year, ymd.month, ymd.day);
}
```

Boost via BH: boost-foreach/

```
#include <Rcpp.h>
#include <boost/foreach.hpp>
using namespace Rcpp;
// [[Rcpp::depends(BH)]]

// the C-style upper-case macro name is a bit ugly
#define foreach BOOST_FOREACH

// [[Rcpp::export]]
NumericVector square( NumericVector x ) {

    // elem is a reference to each element in x
    // we can re-assign to these elements as well
    foreach( double& elem, x ) {
        elem = elem*elem;
    }
    return x;
}
```

Boost via BH: `boost-regular-expressions/`

NB: Use `Sys.setenv("PKG_LIBS"="-lboost_regex")`

```cpp
// boost.org/doc/libs/1_53_0/libs/regex/example/snippets/credit_card_example.cpp
#include <Rcpp.h>
#include <string>
#include <boost/regex.hpp>

bool validate_card_format(const std::string& s) {
    static const boost::regex e("(\\d{4}[- ]){3}\\d{4}");
    return boost::regex_match(s, e);
}

// [[Rcpp::export]]
std::vector<bool> regexDemo(std::vector<std::string> s) {
    int n = s.size();
    std::vector<bool> v(n);
    for (int i=0; i<n; i++)
        v[i]  = validate_card_format(s[i]);
    return v;
}
```

XPtr

Passing an XPtr: `passing-cpp-function-pointers/`

Consider two functions modifying a given Armadillo vector:

```cpp
// [[Rcpp::depends(RcppArmadillo)]]
#include <RcppArmadillo.h>

using namespace arma;
using namespace Rcpp;

vec fun1_cpp(const vec& x) {    // a first function
    vec y = x + x;
    return (y);
}

vec fun2_cpp(const vec& x) {    // and a second function
    vec y = 10*x;
    return (y);
}
```

Passing an XPtr: `passing-cpp-function-pointers/`

Using `typedef` to declare `funcPtr` as an interface to a function taking and returning a vector — and defining a function returning a function pointer given a string argument

```
typedef vec (*funcPtr)(const vec& x);

// [[Rcpp::export]]
XPtr<funcPtr> putFunPtrInXPtr(std::string fstr) {
    if (fstr == "fun1")
        return(XPtr<funcPtr>(new funcPtr(&fun1_cpp)));
    else if (fstr == "fun2")
        return(XPtr<funcPtr>(new funcPtr(&fun2_cpp)));
    else
        // runtime err.: NULL no XPtr
        return XPtr<funcPtr>(R_NilValue);
}
```

Passing an XPtr: `passing-cpp-function-pointers/`

Next we create a function calling the supplied function on a given vector by ‘unpacking’ the function pointer:

```
// [[Rcpp::export]]
vec callViaXPtr(const vec x, SEXP xpsexp) {
    XPtr<funcPtr> xpfun(xpsexp);
    funcPtr fun = *xpfun;
    vec y = fun(x);
    return (y);
}
```

Passing an XPtr: passing-cpp-function-pointers/

Putting it together:

```r
# this gets us a function
fun <- putFunPtrInXPtr("fun1")
# and pass it down to C++ to
# have it applied on given vector
callViaXPtr(1:4, fun)
```

This mechanism is generic and can be used for objective functions, gradients, samplers, ... to operate at C++ speed on user-supplied C++ functions.

Plugins

Plugin support in Rcpp

```
# setup plugins environment
.plugins <- new.env()
# built-in C++11 plugin
.plugins[["cpp11"]] <- function() {
    if (getRversion() >= "3.1")
        list(env = list(USE_CXX1X = "yes"))
    else if (.Platform$OS.type == "windows")
        list(env = list(PKG_CXXFLAGS = "-std=c++0x"))
    else
        list(env = list(PKG_CXXFLAGS ="-std=c++11"))
}
# built-in OpenMP++11 plugin
.plugins[["openmp"]] <- function() {
    list(env = list(PKG_CXXFLAGS="-fopenmp", PKG_LIBS="-fopenmp"))
}

# register a plugin
registerPlugin <- function(name, plugin) {
    .plugins[[name]] <- plugin
}
```

C++11 AUTO: first-steps-with-C++11/

```cpp
#include <Rcpp.h>

// Enable C++11 via this plugin
// [[Rcpp::plugins(”cpp11”)]]

// [[Rcpp::export]]
int useAuto() {
    auto val = 42;      // val will be of type int
    return val;
}
```

C++11 initlist: `first-steps-with-C++11/`

```
#include <Rcpp.h>

// [[Rcpp::plugins(”cpp11”)]]

// [[Rcpp::export]]
std::vector<std::string> useInitLists() {
    std::vector<std::string> vec =
        {”larry”, ”curly”, ”moe”};
    return vec;
}
```

C++11 RANGE: first-steps-with-C++11/

```
#include <Rcpp.h>

// [[Rcpp::plugins(”cpp11”)]]

// [[Rcpp::export]]
int simpleProd(std::vector<int> vec) {
    int prod = 1;
    for (int &x : vec) {       // loop over all values of vec
       prod *= x;              // access each elem., comp. proc
    }
    return prod;
}
```

C++11 LAMBDA: simple-lambda-func-c++11/

```
#include <Rcpp.h>

// [[Rcpp::plugins(”cpp11”)]]

// [[Rcpp::export]]
std::vector<double>
transformEx(const std::vector<double>& x) {
    std::vector<double> y(x.size());
    std::transform(x.begin(), x.end(), y.begin(),
                   [](double x) { return x*x; } );
    return y;
}
```

OpenMP: using-rcppprogress/

We start we with (somewhat boring/made-up) slow double-loop:

```
#include <Rcpp.h>


// [[Rcpp::export]]
double long_computation(int nb) {
    double sum = 0;
    for (int i = 0; i < nb; ++i) {
        for (int j = 0; j < nb; ++j) {
            sum += R::dlnorm(i+j, 0.0, 1.0, 0);
        }
    }
    return sum + nb;
}
```

OpenMP: using-rcppprogress/

```
// [[Rcpp::plugins("openmp")]]
#include <Rcpp.h>

// [[Rcpp::export]]
double long_computation_omp(int nb, int threads=1) {
#ifdef _OPENMP
    if (threads > 0) omp_set_num_threads( threads );
    REprintf("Number of threads=%i\n", omp_get_max_threads());
#endif

    double sum = 0;
#pragma omp parallel for schedule(dynamic)
    for (int i = 0; i < nb; ++i) {
        double thread_sum = 0;
        for (int j = 0; j < nb; ++j) {
            thread_sum += R::dlnorm(i+j, 0.0, 1.0, 0);
        }
        sum += thread_sum;
    }
    return sum + nb;
}
```

OpenMP: using-rcppprogress/

Even on my laptop gains can be seen:

```
R> sourceCpp("code/openmpEx.cpp")
R> system.time(long_computation(1e4))
   user  system elapsed
 22.436   0.000  22.432
R> system.time(long_computation_omp(1e4,4))
Number of threads=4
   user  system elapsed
 25.432   0.076   7.046
R>
```

RcppParallel

Parallel Matrix Transform: `parallel-matrix-transform/`

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

#include <cmath>
#include <algorithm>

// [[Rcpp::export]]
NumericMatrix matrixSqrt(NumericMatrix orig) {
  // allocate the matrix we will return
  NumericMatrix mat(orig.nrow(), orig.ncol());
  // transform it
  std::transform(orig.begin(), orig.end(), mat.begin(), ::sqrt);
  // return the new matrix
  return mat;
}
```

Parallel Matrix Transform: parallel-matrix-transform/

```
// [[Rcpp::depends(RcppParallel)]]
#include <RcppParallel.h>
using namespace RcppParallel;

struct SquareRoot : public Worker {
   const RMatrix<double> input;      // source matrix

   RMatrix<double> output;           // destination matrix

   // initialize with source and destination
   SquareRoot(const NumericMatrix input, NumericMatrix output)
      : input(input), output(output) {}

   // take the square root of the range of elements requested
   void operator()(std::size_t begin, std::size_t end) {
      std::transform(input.begin() + begin,
                     input.begin() + end,
                     output.begin() + begin,
                     ::sqrt);
   }
};
```

Parallel Matrix Transform: `parallel-matrix-transform/`

```cpp
// [[Rcpp::export]]
NumericMatrix parallelMatrixSqrt(NumericMatrix x) {

  // allocate the output matrix
  NumericMatrix output(x.nrow(), x.ncol());

  // SquareRoot functor (pass input and output matrixes)
  SquareRoot squareRoot(x, output);

  // call parallelFor to do the work
  parallelFor(0, x.length(), squareRoot);

  // return the output matrix
  return output;
}
```