Sharing early experiences of programming with Rcpp

Tomasz Woźniak University of Melbourne

Macroeconometrics

Rcpp package

Rcpp package facilitates the application of cpp for computations in R.

It provides interface for communication between R and cpp simplifying it.

It is much easier to benefit from the best of the two worlds:

cpp code is compiled facilitating quick computations:
perfect for writing functions

R code is interpreted and dynamic:
perfect for data analysis using functions written in cpp

Rcpp package

In this presentation we are focusing on applications to Bayesian estimation that relies on:

elements of programming: functions, loops, etc.

compatible object types: scalars, vectors, matrices, lists, etc.

linear algebra: using library Armadillo

random number generators: fast and reproducible using Rcpp

Rcpp is an R package providing the interface, object type compatibility, ..., and vectorised random number generators

Armadillo is a cpp library for linear algebra with fantastic documentation online

RcppArmadillo is an R package providing simplified interface with Armadillo and providing compatibility with its object types

Rcpp package: learning resources

book: Eddelbuettel (2013) Seamless R and C++ Integration with Rcpp

bookdown: Tsuda (2020) Rcpp for everyone

vignettes: for packages Rcpp and RcppArmadillo (published in JSS, CSDA, and TAS)

datacamp course: Optimizing R Code with Rcpp by Romain François

online resources: RcppGallery, Armadillo library documentation, stackoverflow.com

A simple example

Developing a function in a .cpp file

nicetry.cpp

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

using namespace Rcpp;
using namespace arma;

// [[Rcpp::export]]
vec nicetry (mat y, mat x) {
  vec beta_hat = solve(x.t()*x, x.t()*y);
  return beta_hat;
}

/*** R
x = cbind(rep(1,5),1:5)
y = x %*% c(1,2) + rnorm(5)
nicetry(y, x)
solve(crossprod(x), crossprod(x,y))
*/
```

Using a function from a .cpp file

nicetry.cpp

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

using namespace Rcpp;
using namespace arma;

// [[Rcpp::export]]
vec nicetry (mat y, mat x) {
  vec beta_hat = solve(x.t()*x, x.t()*y);
  return beta_hat;
}
```

linreg.R

```
library(Rcpp)
sourceCpp("nicetry.cpp")

x           = cbind(rep(1,5),1:5)
y           = x %*% c(1,2) + rnorm(5)
beta.hat    = nicetry(y, x)
```

Simulation Smoother

Simulation smoother

Sample random draws from the multivariate normal distribution:

$$\mathcal{N}_T\left(\Omega^{-1}\alpha, \Omega^{-1}\right)$$

Ω - $T \times T$ precision matrix that is band- or tri-diagonal

α - $T \times 1$ location vector

A random draw is computed by

$$L^{-1\prime}\left(L^{-1}\alpha + \epsilon\right)$$

$L = \mathbf{chol}(\Omega)$ - is lower-triangular

ϵ - T-vector with independent standard normal draws

R: simulation smoother #1

rmvnorm_solve.R

```
rmvnorm_solve = function(n, precision, location){
  T             = dim(precision)[1]
  precision.chol.inv    = solve(t(chol(precision)))

  epsilon       = matrix(rnorm(n*T), ncol=n)
  draw          = t(precision.chol.inv) %*%
                   (matrix(rep(precision.chol.inv%*%
                      location,n), ncol=n) + epsilon)
  return(draw)
}
```

R: simulation smoother #2

rmvnorm_bandchol.R

```
library(mgcv)

rmvnorm_bandchol = function(n, precision, location){
  T            = dim(precision)[1]
  precision.L = t(bandchol(precision))

  epsilon      = matrix(rnorm(n*T), ncol=n)
  a            = forwardsolve(precision.L, location)
  draw         = backsolve(t(precision.L),
                           matrix(rep(a,n), ncol=n) + epsilon)

  return(draw)
}
```

R: simulation smoother #3

rmvnorm_trichol.R

```
library(mgcv)

rmvnorm_trichol = function(n, precision, location){
  T             = dim(precision)[1]
  lead.diag     = diag(precision)
  sub.diag      = sdiag(precision, -1)

  precision.chol     = trichol(ld = lead.diag, sd=sub.diag)
  precision.L        = diag(precision.chol$ld)
  sdiag(precision.L,-1) = precision.chol$sd

  epsilon       = matrix(rnorm(n*T), ncol=n)
  a             = forwardsolve(precision.L, location)
  draw          = backsolve(t(precision.L),
                            matrix(rep(a,n), ncol=n)
                            + epsilon)
  return(draw)
}
```

rmvnorm_arma_inv.cpp

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

using namespace Rcpp;
using namespace arma;

// [[Rcpp::export]]
mat rmvnorm_arma_inv (int n, mat precision, vec location){
  int T = precision.n_rows;

  mat epsilon(T, n);
  for (int i=0; i<n; i++){
    epsilon.col(i) = as<vec>(rnorm(T));
  }

  mat location_matrix(T, n, fill::zeros);
  location_matrix.each_col() += location;
  mat precision_chol_inv = trans(inv(trimatu(chol(precision))));
  mat draw    = trans(precision_chol_inv) *
                (precision_chol_inv * location_matrix
                 + epsilon);
  return draw;
}
```

rmvnorm_arma_solve.cpp

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

using namespace Rcpp;
using namespace arma;

// [[Rcpp::export]]
mat rmvnorm_arma_solve(int n, mat precision, vec location){
  int T = precision.n_rows;

  mat epsilon(T, n);
  for (int i=0; i<n; i++){
    epsilon.col(i) = as<vec>(rnorm(T));
  }

  mat location_matrix(T, n);
  location_matrix.each_col() = location;
  mat precision_chol  = trimatu(arma::chol(precision));
  mat draw            = solve(precision_chol,
                          solve(trans(precision_chol),
                          location_matrix) + epsilon);
  return draw;
}
```

rmvnorm_arma_stochvol.cpp

```
// [[Rcpp::export]]
mat rmvnorm_arma_stochvol(int n, mat precision, vec location){
  // This algorithm requires the precision matrix
  int T = precision.n_rows;
  vec     precision_diag    = precision.diag();
  double  precision_offdiag = precision(1,0);

  List  precision_chol  = cholesky_tridiagonal(
                precision_diag, precision_offdiag);
  vec   aa              = forward_algorithm(
                  precision_chol["chol_diag"],
                  precision_chol["chol_offdiag"],
                  location);
  mat draw(T, n);
  vec epsilon;
  for (int i=0; i<n; i++){
    epsilon     = rnorm(T);
    draw.col(i) = backward_algorithm(precision_chol["chol_diag"],
                                     precision_chol["chol_offdiag"],
                                     aa + epsilon);
  }
  return draw;
}
```

Simulation smoother

SimulationSmoother.R

```
library(mgcv)
library(Rcpp)
library(microbenchmark)

source("rmvnorm_trichol.R")
source("rmvnorm_bandchol.R")
source("rmvnorm_solve.R")
sourceCpp("rmvnorm_arma_inv.cpp")
sourceCpp("rmvnorm_arma_solve.cpp")
sourceCpp("rmvnorm_arma_stochvol.cpp")

set.seed(12345)
n      = 100
T      = 250
s      = rgamma(1, shape=10, scale=10)
precision = rgamma(1, shape=10, scale=10)*diag(T) + 2*s*diag(T)
sdiag(precision, 1) = -s
sdiag(precision, -1) = -s
location = as.matrix(rnorm(T))
```

Simulation smoother

SimulationSmoother.R

```
microbenchmark(
  R.solve   = rmvnorm_solve(n, precision, location),
  R.band    = rmvnorm_bandchol(n, precision, location),
  R.tridiag = rmvnorm_trichol(n, precision, location),
  cpp.inv = rmvnorm_arma_inv(n, precision, location),
  cpp.sol = rmvnorm_arma_solve(n, precision, location),
  cpp.sto = rmvnorm_arma_stochvol(n, precision, location),
  check  = "equal",
  setup  = set.seed(123)
)
```

```
Unit: milliseconds
      expr       min        lq      mean    median        uq        max neval
   R.solve 15.012696 16.022046 19.049814 16.741754 21.296232  41.665376   100
    R.band  5.521327  6.298791 13.280026  6.719498  9.619501 117.470313   100
 R.tridiag  3.674740  4.004751  8.079238  4.449233  4.916731 178.783048   100
   cpp.inv 10.622570 11.113882 12.171498 11.594380 12.052189  21.046931   100
   cpp.sol  2.024739  2.183631  2.789788  2.808968  2.971318   6.256791   100
   cpp.sto  1.556249  1.621727  1.875903  1.882130  1.988038   3.034821   100
```

Gibbs sampler for a local-level model

Gibbs sampler for a local-level model

The model for a unit-root non-stationary variable:

$$y_t = \mu_t + \epsilon_t \qquad \epsilon_t \sim \mathcal{N}\left(0, \sigma^2\right)$$

$$\mu_t = \mu_{t-1} + m_t \qquad m_t \sim \mathcal{N}\left(0, \sigma_m^2\right)$$

Gibbs sampler:

$$\mu | y, \mu_0, \sigma^2, \sigma_n^2 \sim \mathcal{N}_T\left(\overline{V}^{-1}\overline{\mu}, \overline{V}^{-1}\right)$$

$$\overline{V}^{-1} = \sigma^{-2} I_T + \sigma_m^{-2} H'H$$

$$\overline{\mu} = \sigma^{-2} y + \sigma_m^{-2} \mu_0 e_1$$

$$\mu_0 | y, \mu_1, \sigma_m,^2 \sim \mathcal{N}\left(\left(\sigma_m^{-2} + \underline{v}^{-1}\right)^{-1}, \left(\sigma_m^{-2} + \underline{v}^{-1}\right)^{-1} \sigma_m^{-2} \mu_1\right)$$

$$\sigma^2 | y, \mu \sim \mathcal{IG}2\left(\underline{s} + (y - \mu)'(y - \mu), \underline{\nu} + T\right)$$

$$\sigma_m^2 | \mu, \mu_0 \sim \mathcal{IG}2\left(\underline{s} + (H\mu - \mu_0 e_1)'(H\mu - \mu_0 e_1), \underline{\nu} + T\right)$$

LL_arma_solve.cpp - part 1

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

using namespace Rcpp;
using namespace arma;

// [[Rcpp::export]]
List LL_arma_solve(
    vec y,
    int S = 10,
    Nullable<List> starting_values = R_NilValue,
    NumericVector  Hyper = NumericVector::create(10,1,3)) {

  vec hyper = as<vec>(Hyper);
  int T = y.n_rows;

  vec    aux_mu(T, fill::zeros);
  double aux_mu0      = 0;
  double aux_sigma2   = 1;
  double aux_sigma2m  = 1;
```

LL_arma_solve.cpp - part 2

```
if (starting_values.isNotNull()){
  List Starting_values(starting_values);
  aux_mu               = as<vec>(Starting_values["mu"]);
  aux_mu0              = Starting_values["mu0"];
  aux_sigma2           = Starting_values["sigma2"];
  aux_sigma2m          = Starting_values["sigma2"];
}

mat posterior_mu(T, S);
vec posterior_mu0(S);
vec posterior_sigma2(S);
vec posterior_sigma2m(S);

mat H(T, T, fill::eye);
H.diag(-1) += -1;
mat HH = H.t() * H;
mat IT(T, T, fill::eye);
```

LL_arma_solve.cpp - part 3

```
for (int s=0; s<S; s++){
  double vm      = 1/((1/hyper[0]) + (1/aux_sigma2m));
  aux_mu0        = R::rnorm((vm*aux_mu[0])/aux_sigma2m, sqrt(vm));

  double res_ss = sum(pow(y - aux_mu,2));
  aux_sigma2    = (hyper[1] + res_ss)/R::rchisq(hyper[2] + T);

  vec mu0_vec(1, fill::value(aux_mu0));
  double mu_ss  = sum(pow(diff(join_cols(mu0_vec,aux_mu)), 2));
  aux_sigma2m   = (hyper[1] + mu_ss)/R::rchisq(hyper[2] + T);

  mat precision = IT/aux_sigma2 + HH/aux_sigma2m;
  vec location  = y/aux_sigma2 + aux_mu0*IT.col(0)/aux_sigma2m;
  mat precision_chol  = trimatu(chol(precision));
  vec epsilon         = rnorm(T);
  aux_mu              = solve(precision_chol,
                          solve(trans(precision_chol),
                            location) + epsilon);
  posterior_mu.col(s)      = aux_mu;
  posterior_mu0(s)         = aux_mu0;
  posterior_sigma2(s)      = aux_sigma2;
  posterior_sigma2m(s)     = aux_sigma2m;
}
```

LL_arma_solve.cpp - part 4

```
  List last_draw;
  last_draw["mu"]        = aux_mu;
  last_draw["mu0"]       = aux_mu0;
  last_draw["sigma2"]    = aux_sigma2;
  last_draw["sigma2m"]   = aux_sigma2m;

  List posterior;
  posterior["mu"]        = posterior_mu;
  posterior["mu0"]       = posterior_mu0;
  posterior["sigma2"]    = posterior_sigma2;
  posterior["sigma2m"]   = posterior_sigma2m;

  List output;
  output["last.draw"]    = last_draw;
  output["posterior"]    = posterior;

  return output;
}
```

Gibbs sampler for a local-level model

LL.R

```
microbenchmark(R.not     = LL_nothing_special(y250, S),
               R.ban     = LL_band_precision(y250, S),
               R.tri     = LL_tridiag_precision(y250, S),
               cpp.inv   = LL_arma_inv(y250, S),
               cpp.sol   = LL_arma_solve(y250, S),
               cpp.sto   = LL_arma_stochvol(y250, S),
               setup  = set.seed(123)
               )
```

```
Unit: milliseconds
    expr        min         lq       mean     median         uq       max neval
   R.not 127.266986 132.328077 155.005314 138.435266 154.022634 327.33741   100
   R.ban  46.167909  49.679159  95.663136  55.453045 159.034735 324.58272   100
   R.tri  19.089019  22.110003  41.142920  26.470268  29.056582 150.71913   100
 cpp.inv  10.580124  11.180750  12.486657  11.937587  12.845465  26.01354   100
 cpp.sol  10.542156  11.145461  12.291518  11.682092  12.380952  27.42871   100
 cpp.sto   8.177979   8.542281   9.554215   9.115295   9.739227  19.19876   100
```

What's next?

Rewrite all your code in Rcpp!