Computational Statistics II

Unit A.2: Rcpp & RcppArmadillo

Tommaso Rigon

University of Milano-Bicocca

Ph.D. in Economics and Statistics

Unit A.2

Main concepts

- Introduction to **Rcpp** & **RcppArmadillo**
- Some examples and comparisons with **R**
- The Gaussian model: Gibbs sampling using **R** and C++

- Associated **R** code: `https://tommasorigon.github.io/CompStat/exe/un_A2.html`

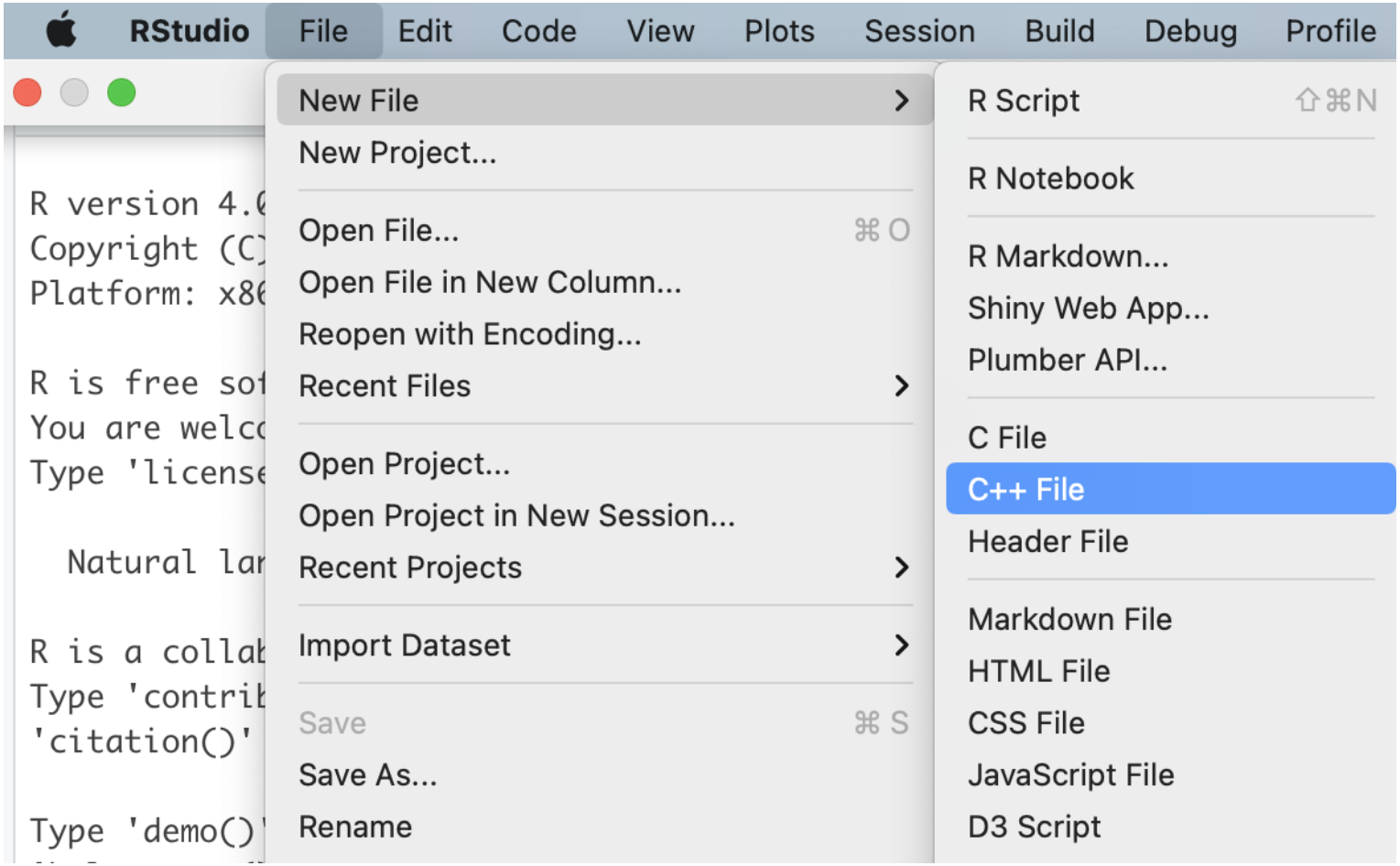
Main references

- Eddelbuettel, D. and Balamuta, J. J. (2018). Extending R with C++: A Brief Introduction to Rcpp. *The American Statistician*, **72**(1), 28–36.
- Eddelbuettel, D., and Sanderson, C. (2014). RcppArmadillo: Accelerating R with high-performance C++ linear algebra. *Computational Statistics and Data Analysis*, **71**, 1054–1063.

Introduction

- The **Rcpp** package **simplifies the interface** between **R** and C++.
- The package **RcppArmadillo** extends **Rcpp** and simplifies the interface between **R** and armadillo, which is a "high quality **linear algebra** library for the C++ language, aiming towards a good balance between speed and ease of use".
- The main advantage is that C++ code is usually **much faster than R** (and python), especially in non-vectorized settings.
- It is very hard to be faster than **Rcpp**, unless you code is indeed written in C++.

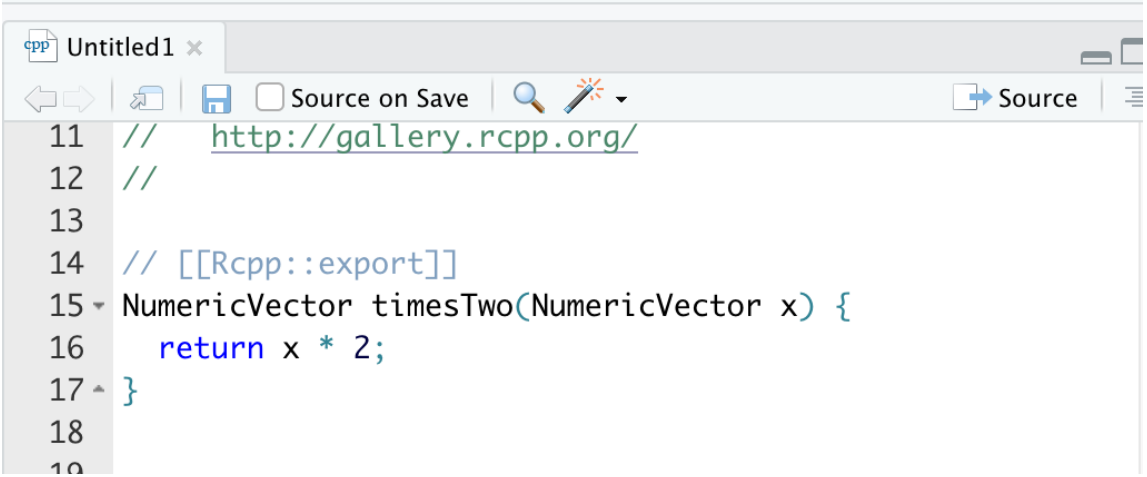
Basic usage



- Nowadays, both packages (**Rcpp** and **RcppArmadillo**) are very well integrated within RStudio.

Basic usage

- Provided you installed everything correctly (refer to the the Markdown A.2 available in the course website), the typical **coding pipeline** is straightforward.
- First, create an empty file, say `foo.cpp`, containing the C++ code.
- Save the C++ file and **compile it** using the `sourceCpp` function. Alternatively, you can press the "source" button if you are using RStudio.
- Use the functions contained in the C++ file within **R** as usual. The functions will appear in the environment.



The sum function in **RcppArmadillo**

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

// [[Rcpp::export]]
double arma_sum(vec x){
  double sum = 0;
  int n = x.n_elem; // Length of the vector x
  for(int i=0; i < n; i++){
    sum += x[i]; // Shorthand for: sum = sum + x[i];
  }
  return(sum);
}
```

```r
sourceCpp("../cpp/sum.cpp")
x <- c(10, 20, 5, 30, 21, 78, pi, exp(7))
arma_sum(x) # sum of the vector x
# [1] 1263.775
sum(x) # sum of the vector x - usual command
# [1] 1263.775
```

Example 1: Euclidean distance

- The **R** code is typically **slow** in presence of (nested) `for` loops.
- We are given a matrix $\boldsymbol{X}$ of dimension $n \times p$, whose rows are $x_i = (x_{i1}, \ldots, x_{ip})^T$.
- We are interested in computing the matrix of euclidean distances D of dimension $n \times n$ whose entries are equal to

$$d_{ii'} = \sqrt{\sum_{j=1}^{p} (x_{ij} - x_{i'j})^2}, \qquad i, i' \in \{1, ..., n\}.$$

```
R_dist <- function(X) {
  n <- nrow(X)
  D <- matrix(0, n, n) # Pre-allocate the output
  for (i in 1:n) {
    for (k in 1:i) {
      D[i, k] <- D[k, i] <- sqrt(sum((X[i, ] - X[k, ])^2))
    }
  }
  D
}
```

Example 1: Euclidean distance

- The corresponding **RcppArmadillo** implementation is quite simple as well.

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

// [[Rcpp::export]]
mat arma_dist(const mat& X){
  int n = X.n_rows;
  mat D(n, n, fill::zeros); // Allocate a matrix of dimension n x n
  for (int i = 0; i < n; i++) {
    for(int k = 0; k < i; k++){
      D(i, k) = sqrt(sum(pow(X.row(i) - X.row(k), 2)));
      D(k, i) = D(i, k);
    }
  }
  return D;
}
```

Example 1: benchmark

```r
X <- as.matrix(USArrests) # Example dataset

benchmark(
  arma_dist = arma_dist(X), # Armadillo implementation
  R_dist = R_dist(X), # Naive R implementation
  dist = as.matrix(dist(X)), # Built-in R function (C++)
  columns = c("test", "replications", "elapsed", "relative"),
  replications = 1000
)

#        test replications elapsed relative
# 1 arma_dist         1000   0.015    1.000
# 3      dist         1000   0.080    5.333
# 2    R_dist         1000   2.445  163.000
```

- Let us use the `USArrests` dataset for a quick **benchmark**.
- The **RcppArmadillo** implementation is about **150 times faster** than the naive **R** version, due to the presence of nested for loops.
- Actually, the **RcppArmadillo** version is slightly faster the `dist` built-in **R** function!

Example 2: linear models

- **R** code is not necessarily slower than Armadillo when linear algebra is involved.
- Suppose we are interested in obtaining the least squares estimate $\hat{\beta}$ from the design matrix $\boldsymbol{X}$ and the response $\boldsymbol{y}$, namely $\hat{\beta} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y}$.
- In first place, let us compare two slightly different **R** implementations.
- As a rule of thumb, do not invert matrices if the actual goal is solving linear systems.

```
# Using matrix multiplication commands
lm_coef1 <- function(X, y) {
  solve(t(X) %*% X) %*% t(X) %*% y
}

# Better (no matrix inversion!) and faster implementation
lm_coef2 <- function(X, y) {
  solve(crossprod(X), crossprod(X, y))
}
```

Example 2: linear models

- The `solve` function here can be used directly on the objects $\boldsymbol{X}$ and $\boldsymbol{y}$.

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

// [[Rcpp::export]]
vec lm_coef3(const mat& X, const vec& y) {
  vec coef = solve(X, y);
  return(coef);
}
```

```
set.seed(123)
X <- cbind(1, rnorm(10^4))
y <- rowSums(X) + rnorm(10^4)

cbind(lm_coef1(X, y), lm_coef2(X, y), lm_coef3(X, y)) # Same results

#           [,1]      [,2]      [,3]
# [1,] 0.9909079 0.9909079 0.9909079
# [2,] 1.0060394 1.0060394 1.0060394
```

Example 2: benchmark

```
benchmark(R_matrix_inv = lm_coef1(X, y),
  R_no_matrix_inv  = lm_coef2(X, y),
  Rcpp = lm_coef3(X, y),
  lm = coef(lm(y ~ X, data = cars)),
  columns = c("test", "replications", "elapsed", "relative"),
  replications = 1000
)

#              test replications elapsed relative
# 4              lm         1000   2.487   25.378
# 1    R_matrix_inv         1000   0.365    3.724
# 2 R_no_matrix_inv         1000   0.098    1.000
# 3            Rcpp         1000   0.141    1.439
```

- In this case, the **RcppArmadillo** implementation is approximately as fast as the **R** version.
- Indeed, the "difficult" part (i.e. solution of the linear system) in all cases is handled by well-optimized C routines.
- The usual `lm` **R** functions is slower, but it is calculating many additional quantities.

Gibbs sampling (recap)

- Recall that $\pi(\boldsymbol{\theta} \mid \boldsymbol{X})$ denotes the posterior distribution of $\boldsymbol{\theta} \in \Theta \subseteq \mathbb{R}^p$ given the data.
- Let us partition the parameter vector $\boldsymbol{\theta} = (\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_B)$ into B blocks of parameters. Sometimes we will have as many blocks as parameters, so that $\boldsymbol{\theta} = (\theta_1, \ldots, \theta_p)$.
- Let $\pi(\boldsymbol{\theta}_b \mid -)$ be the so-called **full-conditional** of $\boldsymbol{\theta}_b$, that is

$$\pi(\boldsymbol{\theta}_b \mid -) = \pi(\boldsymbol{\theta}_b \mid \boldsymbol{X}, \boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_{b-1}, \boldsymbol{\theta}_{b+1}, \ldots, \boldsymbol{\theta}_B), \quad b = 1, \ldots, B,$$

  namely the conditional distribution of $\boldsymbol{\theta}_b$ given the data and the other parameters.
- Repeatedly sampling $\boldsymbol{\theta}_b$, for $b = 1, \ldots, B$, from the corresponding full conditionals leads to a MCMC algorithm targeting the posterior distribution $\pi(\boldsymbol{\theta} \mid \boldsymbol{X})$.
- In the next units (B.1, C.1, C.2), we will see extensions of this basic strategy, but at the moment we will focus on its **implementation**.

Example 3: conditionally-conjugate Gaussian model

- Let us assume the observations $(x_1, \ldots, x_n)$ are draws from

$$(x_i \mid \mu, \sigma^2) \overset{\text{iid}}{\sim} \mathsf{N}(\mu, \sigma^2), \qquad i = 1, \ldots, n.$$

  with independent priors $\mu \sim \mathsf{N}(\mu_\mu, \sigma^2_\mu)$ and $\sigma^{-2} \sim \mathsf{Ga}(a_\sigma, b_\sigma)$.

- The full conditional distribution for the **mean** μ is:

$$(\mu \mid -) \sim \mathsf{N}\left(\mu_n, \sigma^2_n\right), \qquad \mu_n = \sigma^2_n \left(\frac{\mu_\mu}{\sigma^2_\mu} + \frac{1}{\sigma^2} \sum_{i=1}^{n} x_i \right), \quad \sigma^2_n = \left(\frac{n}{\sigma^2} + \frac{1}{\sigma^2_\mu} \right)^{-1}.$$

- The full conditional distribution for the **precision** σ^{-2} is:

$$(\sigma^{-2} \mid -) \sim \mathsf{Ga}\left(a_n, b_n\right), \qquad a_n = a_\sigma + n/2, \quad b_n = b_\sigma + \frac{1}{2} \sum_{i=1}^{n} (x_i - \mu)^2.$$

Example 3: implementation in **R**

```r
gibbs_R <- function(x, mu_mu, sigma2_mu, a_sigma, b_sigma, R, burn_in) {
  # Initialization
  n <- length(x); xbar <- mean(x)
  out <- matrix(0, R, 2)
  # Initial values for mu and sigma
  sigma2 <- var(x); mu <- xbar
  for (r in 1:(burn_in + R)) {
    # Sample mu
    sigma2_n <- 1 / (1 / sigma2_mu + n / sigma2)
    mu_n <- sigma2_n * (mu_mu / sigma2_mu + n / sigma2 * xbar)
    mu <- rnorm(1, mu_n, sqrt(sigma2_n))
    # Sample sigma2
    a_n <- a_sigma + 0.5 * n
    b_n <- b_sigma + 0.5 * sum((x - mu)^2)
    sigma2 <- 1 / rgamma(1, a_n, b_n)
    # Store the values after the burn-in period
    if (r > burn_in) {
      out[r - burn_in, ] <- c(mu, sigma2)
    }
  }
  out
}
```

Example 3: implementation in **RcppArmadillo**

```
mat gibbs_arma(vec x, double mu_mu, double sigma2_mu,
double a_sigma, double b_sigma, int R, int burn_in){
  // Initialization
  double n = x.n_elem; double xbar = mean(x);
  mat out(R,2);
  // Initial values for mu and sigma
  double sigma2 = var(x); double mu = mean(x);
  for (int r = 0; r < R + burn_in; r++) {
    // Sample mu
    double sigma2_n = 1.0 / (1.0 / sigma2_mu + n / sigma2);
    double mu_n = sigma2_n * (mu_mu / sigma2_mu + n / sigma2 * xbar);
    mu = rnorm(1, mu_n, sqrt(sigma2_n))[0];
   // Sample sigma2
    double a_n = a_sigma + 0.5 * n;
    double b_n = b_sigma + 0.5 * sum(pow(x - mu, 2));
    sigma2 = 1.0 / rgamma(1, a_n, 1.0 / b_n)[0]; // NOTE: rgamma in Rcpp use scale not rate
    if (r > burn_in) {
      out(r - burn_in, 0) = mu;
      out(r - burn_in, 1) = sigma2;
    }
  }
  return out;
}
```

Example 3: benchmark

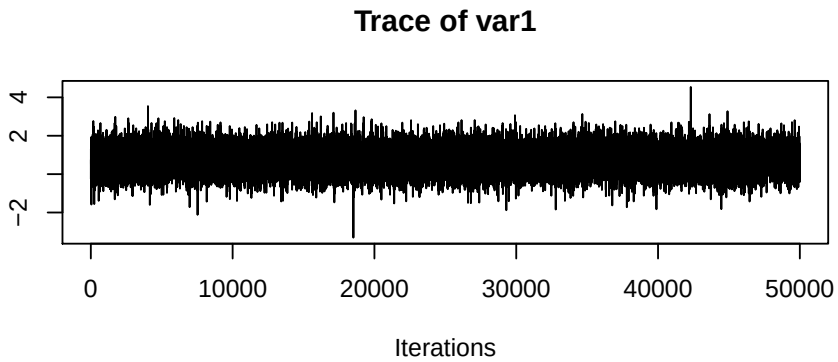
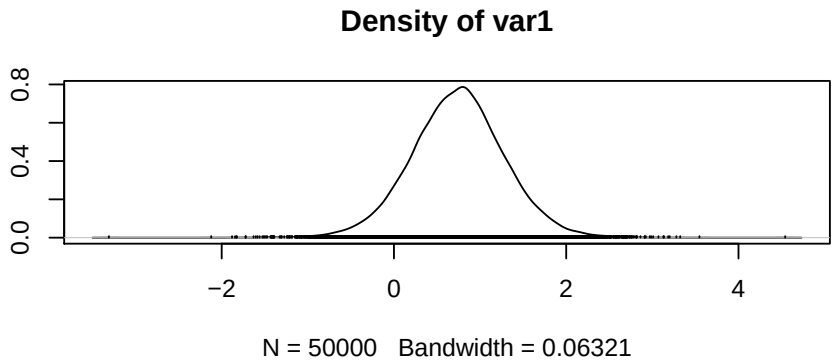
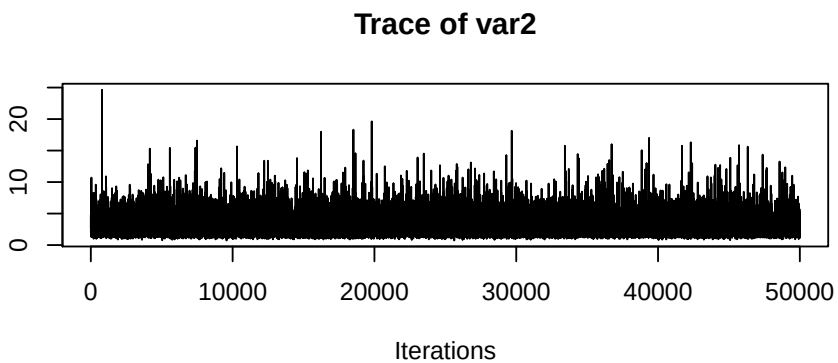
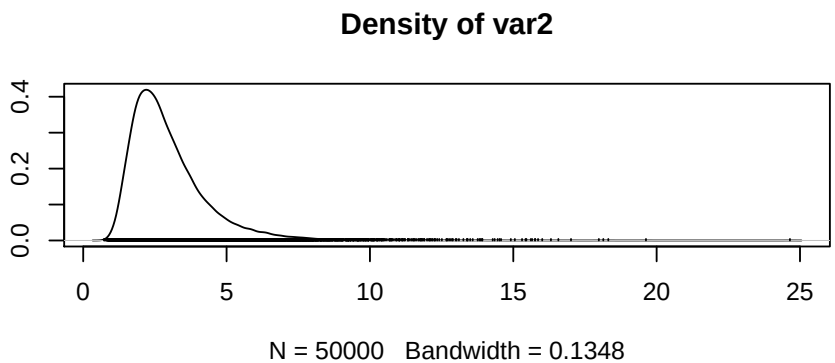
- In this dataset, the **RcppArmadillo** implementation is about **25 times faster** than **R**.

```
data(sleep)
x <- sleep$extra[sleep$group == 1]

library(tictoc) # Library for "timing" the functions
set.seed(123)

# Hyperparameter settings
mu_mu <- 0; sigma2_mu <- 50
a_sigma <- b_sigma <- 2
R <- 50000; burn_in <- 5000

tic()
fitR <- gibbs_R(x, mu_mu, sigma2_mu, a_sigma, b_sigma, R, burn_in)
toc()
# 0.415 sec elapsed

tic()
fitRcpp <- gibbs_arma(x, mu_mu, sigma2_mu, a_sigma, b_sigma, R, burn_in)
toc()
# 0.015 sec elapsed
```

Traceplots and posterior distribution