# A case study in the implementation of a Bayesian linear model using R and / or C++ libraries

## Including speed comparisons (and more to come)

*David Pritchard*

August 3, 2016

# 0 Introduction

As an exercise I decided to implement a Bayesian linear model using the C++ libraries Armadillo and Eigen, and then to interface the C++ code from R via Rcpp. The main purposes of this exercise were, as one might guess from the choice of exercise, to obtain some familiarity with Armadillo, Eigen, and Rcpp.

The reason for choosing the Bayesian linear model as the algorithm that I would try to implement was that (i) at the time of writing this I have a Gibbs sampler in hand that I wish to code in C++ so the skills learned here will be applicable to that problem, and (ii) this particular Gibbs sampler is a very simple algorithm so that consequently the comparison of the implementations in terms of speed is also relatively simple.

There were also some secondary skills that I wanted to practice when doing the exercise. Some of these included:

- Comparison of the speeds of the algorithm implementations
- Compare the speeds of using various implementations of BLAS and LAPACK
- Practice profiling programs in R and in C++

## 0.1 Computer specifications

The speed and profiling results presented in the document below will be highly dependent on the computing environment on which they were obtained. A short summary of the specifications of the computing environment for which the results in this document were obtained are listed below:

- Intel i3 64-bit processor
- 2 hyper-threaded cores
- 4 GB of main memory
- 32 KB L1 cache
- 256 KB L2 cache
- 3 MB L3 cache
- Ubuntu GNU/Linux 14.04 system

# 1 The Bayesian linear model

The Bayesian linear model and resulting algorithm which I consider follows the definition and presentation of Peter Hoff's *A First Course in Bayesian Statistical Methods*, sections 9.1 - 9.2.

## 1.1 Model definition

Consider the model

$$\boldsymbol{y} = \boldsymbol{X}\boldsymbol{\beta} + \boldsymbol{\epsilon} \tag{1}$$

where

$\boldsymbol{\epsilon} \,|\, \sigma^2 \sim \mathcal{N}(\mathbf{0},\, \sigma^2 I)$

$\sigma^2 \sim \text{inverse-gamma}(\nu_0/2,\, \nu_0\sigma_0^2/2)$

$\boldsymbol{\beta} \sim \mathcal{N}(\boldsymbol{\beta}_0,\, \boldsymbol{\Sigma}_0)$

### 1.2 Full conditional distributions

Then under the model presented in section 1.1, it can be shown that the full conditional distributions are given by

$$\sigma^2 \,|\, \boldsymbol{y}, \boldsymbol{X}, \boldsymbol{\beta} \sim \text{inverse-gamma}\Big( [\nu_0 + n]/2, \;\; \big[\nu_0 \sigma_0^2 + \text{SSR}(\boldsymbol{\beta}\big] \,/2 \Big)$$

$$\boldsymbol{\beta} \,|\, \boldsymbol{y}, \boldsymbol{X}, \boldsymbol{\beta} \sim \mathcal{N}(\boldsymbol{m}, \, \boldsymbol{V})$$

where

$$\text{SSR}(\boldsymbol{\beta}) = (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta})^T (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta})$$

$$\boldsymbol{V} = \Big( \boldsymbol{\Sigma}_0^{-1} + \boldsymbol{X}^T \boldsymbol{X} / \sigma^2 \Big)^{-1}$$

$$\boldsymbol{m} = \boldsymbol{V} \big( \sigma_0^{-1} \boldsymbol{\beta}_0 + \boldsymbol{X}^T \boldsymbol{y} / \sigma^2 \big)$$

Notice that the full conditional distribution for $\boldsymbol{\epsilon}$ is not mentioned - this is because it is completely determined conditional on $(\boldsymbol{y}, \boldsymbol{X}, \boldsymbol{\beta})$. In fact, it is more natural in a Bayesian setting to rewrite (1) as $\boldsymbol{y} \,|\, \boldsymbol{X}\boldsymbol{\beta} \sim \mathcal{N}(\boldsymbol{X}\boldsymbol{\beta}, \sigma^2 I)$, which obviates any mention of $\epsilon$ from the model.

### 1.3 Gibbs sampler algorithm

It follows from section 1.2 that an approximation for the joint posterior distribution $p(\boldsymbol{\beta}, \sigma^2 \,|\, \boldsymbol{y}, \boldsymbol{X})$ can be obtained as follows:

1. update $\boldsymbol{\beta}$:
   (a) compute $\boldsymbol{V}^{(s+1)} = \Big( \boldsymbol{\Sigma}_0^{-1} + \boldsymbol{X}^T \boldsymbol{X} / \sigma^{2(s)} \Big)^{-1}$
   (b) compute $\boldsymbol{m}^{(s+1)} = \boldsymbol{V}^{(s+1)} \Big( \boldsymbol{\Sigma}_0^{-1} \boldsymbol{\beta}_0 + \boldsymbol{X}^T \boldsymbol{y} / \sigma^{2(s)} \Big)$
   (c) sample $\boldsymbol{\beta}^{(s+1)} \sim \mathcal{N}\big( \boldsymbol{m}^{(s+1)}, \, \boldsymbol{V}^{(s+1)} \big)$
2. update $\sigma^2$:
   (a) compute $\text{SSR}(\boldsymbol{\beta}^{(s+1)})$
   (b) sample $\sigma^{2(s+1)} \sim \text{inverse-gamma}\Big( [\nu_0 + n]/2, \;\; \big[\nu_0 \sigma_0^2 + \text{SSR}(\boldsymbol{\beta}^{(s+1)})\big] \,/2 \Big)$

## 2 Progam descriptions

The source code for the following R functions / executable programs is contained in the root directory. The name of each function / program is listed as well as the files for each containing the source code. Note that some of the files are used for more than one implementation.

1. `bayes_lm_r:` an R function
   - `Bayes_LM.R`
   - `Check_Valid_Input.R`
2. `bayes_lm_arma:` a C++-only implementation using the Armadillo library (an executable)
   - `Bayes_LM_Arma.cpp`
   - `Parse_Args.cpp`
   - `Stats_Fcns_Arma.cpp`

3. **bayes_lm_eigen:** a C++-only implementation using the Eigen library (an executable)

    - Bayes_LM_Eigen.R
    - Parse_Args.cpp
    - Stats_Fcns_Eigen.cpp

4. **bayes_lm_rcpp_arma:** an R function internally calling a workhorse C++ function constructed using the Armadillo library

    - Bayes_LM_Rcpp_Arma.R
    - Check_Valid_Input.R
    - Stats_Fcns_Arma.cpp

5. **bayes_lm_rcpp_eigen:** an R function internally calling a workhorse C++ function constructed using the Eigen library

    - Bayes_LM_Rcpp_Eigen.R
    - Check_Valid_Input.R
    - Stats_Fcns_Eigen.cpp

## 2.1 Writing bayes_lm_r

The function signature for **bayes_lm_r** is shown in the following. The interfaces to **bayes_lm_rcpp_arma** and **bayes_lm_rcpp_eigen** are similar.

```
bayes_lm_r <- function(n               = 100L,
                       p               = 15L,
                       nsamp           = 1e4,
                       prop_nonzero    = 0.2,
                       true_beta_sd    = 2,
                       true_sigma      = 2,
                       sd_x            = 2,
                       print_stats     = FALSE,
                       time_sections   = TRUE,
                       write_samples   = FALSE,
                       samples_file_loc = "Samples_R.dat",
                       decomp_method   = "chol") {}
```

### 2.1.1 Brief description

In short the function simulates the data $\boldsymbol{X}, \boldsymbol{y}$, and $\boldsymbol{\beta}$, and then performs a Gibbs sampler for the model described in section 1.1 using the simulated data.

### 2.1.2 Fuller description

In more detail, the function samples an $n \times p$ matrix $\boldsymbol{X}$ with each element $X_{ij}$ independently sampled from a normal distribution with mean 0 and standard deviation as specified by `sd_x`. A $p$-length vector $\boldsymbol{\beta}$ is sampled with `prop_nonzero` proportion of its entries having values sampled from a normal distribution with mean 0 and standard deviation `true_beta_sd`; the remaining values are set to 0. Then $\boldsymbol{y}$ is sampled from a normal distribution with mean $\boldsymbol{X\beta}$ and variance $\sigma^2 I$ where $\sigma$ is specified by `true_sigma`.

A Gibbs sampler with `nsamp` scans is then performed. If specified by `print_stats` then upon completion, a summary of the sample quantiles will be written to the R console, and if specified by `write_samples` then each sample will be written to a file with filename as specified by `samples_file_loc`.

The Gibbs sampler algorithm (described in section 1.3) requires sampling from a multivariate normal distribution; doing so requires performing a matrix decomposition. The function provides the option to use either the Cholesky decomposition or the Eigen decomposition for this part of the algorithm, the choice of which can be specified by `decomp_method`.

The formal argument `time_sections` allows the user the option to track the total time spent in the sampler (i) performing matrix inversions (ii) sampling from the multivariate normal distribution. The algorithm requires one execution of each task per scan.

**`bayes_lm_r`** returns a length-3 vector with values providing the time in the sampler spent performing the inversion, the multivariate normal sampling, and the overall time in the sampler. Note that this overall time includes only the sampler and not any computations performed before or after.

### 2.1.3 Coding observations

This is quite a simple model to code (the model was chosen for this reason!). Outside of checking the function arguments for validity, the function spans about 150 lines of code, including comments and blank lines. R provides all of the tools we need for this particular program and makes life very nice and simple for us.

## 2.2 Writing `bayes_lm_arma`

To invoke run the Bayesian linear model sampler using a C++-only version now of course is done from the command-line. To obtain the default parameter specifications (i.e. the specifications obtained by invoking

```
./bayes_lm_arma
```

one could explicitely specify each parameter by

```
./bayes_lm_arma -n=100  \
                -p=15  \
                -prop_nonzero=0.2  \
                -true_beta_sd=2  \
                -sd_x=2  \
                -nsamp=1000  \
                -print_stats=false  \
                -write_ctime=false  \
                -ctime_file_loc=Comp_Time_Arma.dat  \
                -write_samples=false  \
                -samples_file_loc=Samples_Arma.dat  \
                -decomp_method=c  \
                -set_seed=22
```

*Actually, if left unspecified the seed is generating using the CPU clock so there isn't really a default for `set_seed`. That way running the program without specifiying a seed generates a different sequence every time. The number 22 was arbitrarily chosen here as an example.

### 2.2.1 Overall impression

This was my first exposure to the Armadillo library. I found it to be very intuitive and convenient to work with. The documentation was relatively clear and helpful. If I could offer one suggestion for room for improvement regarding the documentation, I wish that it would provide a little more detail on the underlying algorithms that are being used to perform the provided functionality.

### 2.2.2 Choosing a random number generating method

One decision that has to be made when constructing a sampler in C++ is how to generate the random numbers. While there are several options, such as GNU's random number generation (RNG) library, or the C++ 11 library, I ended up using R's RNG library for the following reasons: (i) I am very familiar with the R RNG interface, and (ii) we are not allowed to make use of the system RNG for R packages submitted to CRAN (which disqualifies the GNU and C++ 11 libraries). The available options and merits of the available RNG libraries is a topic which I will have to investigate in more detail in future, however.

### 2.2.3 Converting from R code to C++

Now that we're not in the friendly confines of the R programming environment, there was some additional work that needed to be performed to achieve the same functionality.

1. The Armadillo library provides a method to randomly populate a matrix with independent standard normal entries, however it does not provide a method to sample from an arbitrary multivariate normal distibution. Consequently I had to write a multivariate normal sampling function - I wrote one using the Cholesky decomposition, and one using the Eigen decompostion. Each of these functions required just a few lines of code.

2. I also had to construct a quantile function. To do so I emulated the quantile function from R under the default formula; this required about a 100 lines of code.

3. Another change that was required going from R to C++ was reading in the function arguments. In R all of the work that is performed parsing the function arguments is taken care of for me by R; in C++ we have to do a little more manual work to read the function arguments (by arguments here I mean command-line arguments). Of course there are some extremely powerful tools provided by the C standard, so thankfully we don't have to do much work from scratch. However there are still some mundane work that need to be performed when parsing the arguments - this required about 150 lines of code. In comparison the R code that I wrote for argument checking was about 75 lines.

4. One thing that confused me about using the R math library for random number generation was how to properly set up the random number stream and set the seed of the stream. In the *Writing R Extensions* manual section 6.3, it says that all calls to RNG routines should be prefaced by a call to `GetRNGstate()`, and that after all the required variates have been generated, `GetRNGstate()` should be called. However, when using the R math library as a standalone library, the random number stream is automatically set up for you, and setting the seed is accomplished through `set_seed` (declared in `Rmath.h`). See the *Writing R Extensions* manual section 6.16 for more details.

### 2.2.4 Compiling C++ code with Armadillo

The source code can be compiled as follows.

```
g++ Bayes_LM_Arma.cpp Parse_Args.cpp Stats_Fcns_Arma.cpp    \
    -DMATHLIB_STANDALONE -Wall -g3 -O3 -lRmath -larmadillo  \
    -o bayes_lm_arma
```

The first three arguments are filenames of the source code that needs to be pulled into the compilation. The argument `-DMATHLIB_STANDALONE` defines the MATHLIB_STANDALONE macro which in turn configures the `Rmath.h` header for use with the Rmath standalone library. `-Wall` tells `gcc` to provide additional warnings for potentially problematic constructs, `-g3` includes debugging information in the resulting binary file, and `-O3` requests optimization from the compiler (note that this optimization flag is very important for heavily templated code such as the Armadillo and Eigen libraries). The arguments -lRmath and -larmadillo tell the compiler the link the files against the R math library and Armadillo library, respectively. The argument `-o bayes_lm_arma` specifies the filename of the resulting executable.

## 2.3 Writing `bayes_lm_eigen`

### 2.3.1 Overall impression

This was the second time that I had any exposure to Eigen. I find it also to be quite intuitive and convenient to work with. I thought the documentation was well-developed and quite thorough. If I had to mention one area that I found a little not to my taste, it was that many of the classes and methods had very verbose names (e.g. `transpose()` vs. `t()`).

### 2.3.2 Converting Armadillo code to Eigen

Converting Armadillo to Eigen was quite straightforward; in many places I just had to change e.g. `arma::mat` to `Eigen::MatrixXd`. There were a few differences, however.

1. Eigen does not have a method to randomly populate a matrix with independent standard normal entries, so I had to create one; this requires just a handful of lines of code.
2. There wasn't a built-in matrix inverse method that leveraged the fact that the matrix that we are inverting is a symmetric postive-definite matrix. Instead, this required me to use the `solve` method for the Cholesky decomposition, where we are solving for the identity matrix. It would seem that a dedicated method for the inverse could potentially be more efficient. The speed comparisons in section 3.2 seem to support this possibility.

### 2.3.3 Compiling C++ code with Eigen

The source code can be compiled as follows.

```
g++ Bayes_LM_Eigen.cpp Parse_Args.cpp Stats_Fcns_Eigen.cpp  \
    -DMATHLIB_STANDALONE -I/usr/include/eigen3/             \
    -I/usr/share/R/include -Wall -g3 -O3 -lR -lRmath        \
    -o bayes_lm_eigen
```

All of the options here have been previously discussed in section 2.2.4. It is worth noting however, that for Eigen (as apposed to Armadillo) we do not link it against a shared library; instead it is statically linked as part of the compilation process.

## 2.4 Writing `bayes_lm_rcpp_arma` and `bayes_lm_rcpp_eigen`

The steps to convert the C++ code using the Armadillo library to a function for use through R were exactly the same as for converting C++ code using the Eigen library, so I have included them in the same section. Any mention of Armadillo can be replaced by Eigen.

### 2.4.1 Overall impression

This was my first time using Rcpp. It is incredible how convenient it makes it to interface C++ code from R.

### 2.4.2 Interfacing Armadillo code through `R`

Essentially all of the pieces were already written at this point, and it was just a matter of putting them all together. Now that we are back in R-land, I'd prefer and am able to do the argument checking in R again. So the user-facing function simply checks the function arguments for validity and does any necessary preprocessing, and then calls an R function pointing to a function in the dynamically loaded library constructed from the C++ code, supplying it with the now sanitized data. The C++ function pointed to by the caller is a thin wrapper constructed by Rcpp, which takes all of the `SEXP` pointers passed to it from R and creates `Rcpp:RObject`s from them, and in turn calls our C++ function, providing these `Rcpp:RObject` objects as the arguments.

To put all of the pieces together then mainly required:

1. Creating the C++ function

(a) Creating a function signature for the C++ function using C++ primatives and `Rcpp::RObject` derivatives

(b) Copy-pasting the body of the original Armadillo code into the function

(c) Creating a return argument for the function, again using C++ primatives and `Rcpp::RObject` derivatives

2. Creating the R wrapper

   (a) Copy-pasting the function signature and argument-checking lines of the original R function

   (b) Compiling the the C++ code, loading the newly created shared library into memory, and obtaining a pointer to it, all through a call to `Rcpp::sourceCpp`

3. Modifying the file containing the statistical helper functions so that it would work with both code being called from R and as part of a standalone C++ program

   (a) The issue that I had was that that the `rnorm` function wasn't declared when compiling the functions with the `Rcpparmadillo.h` header (as opposed to the `Rmath.h` header). In the `RcppArmadillo.h` header the authors map the alias `R::rnorm` to the `rnorm` function, so what I did as a solution was to create an alias of this same name for the case when the file is included as part of a standalone C++ program, and that way I can call the desired function by `R::rnorm` in both cases.

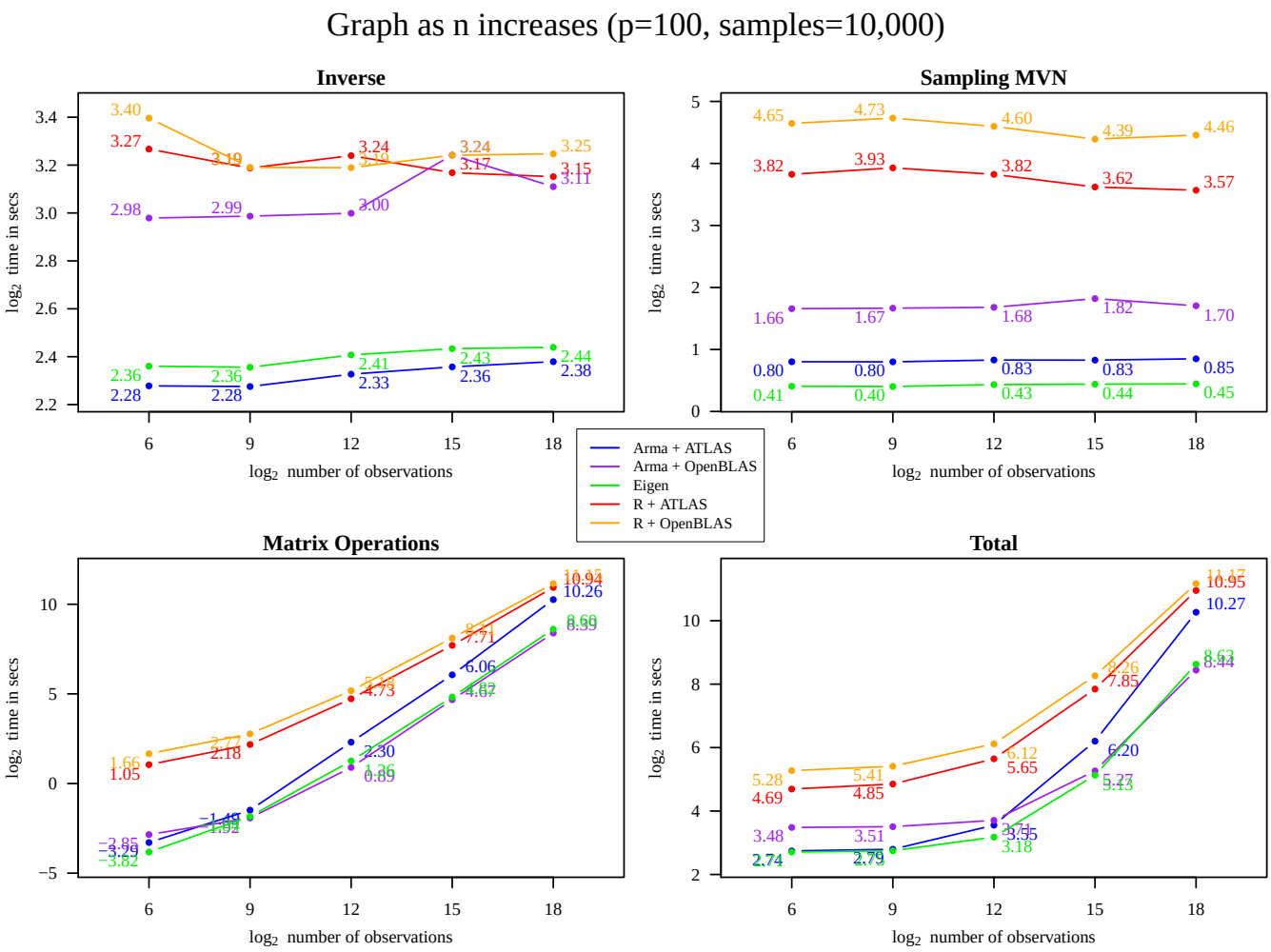
# 3 Comparison of program speeds

The speed in which Bayes linear model sampler could be completed was measured using 5 different software configurations:

| Configuration | Details |
|---|---|
| R + ATLAS | **`bayes_lm_r`** using ATLAS version of both BLAS and LAPACK |
| R + OpenBLAS | **`bayes_lm_r`** using OpenBLAS version of BLAS and reference LAPACK |
| Armadillo + ATLAS | **`bayes_lm_rcpp_arma`** using ATLAS version of both BLAS and LAPACK |
| Armadillo + OpenBLAS | **`bayes_lm_rcpp_arma`** using OpenBLAS version of BLAS and reference LAPACK |
| Eigen | **`bayes_lm_rcpp_eigen`** (does not use BLAS / LAPACK) |

Caveat: to obtain peak efficiencies of ATLAS and OpenBLAS the libraries should be compiled on the local machine, however these tests were run using the ATLAS and OpenBLAS Debian packages.

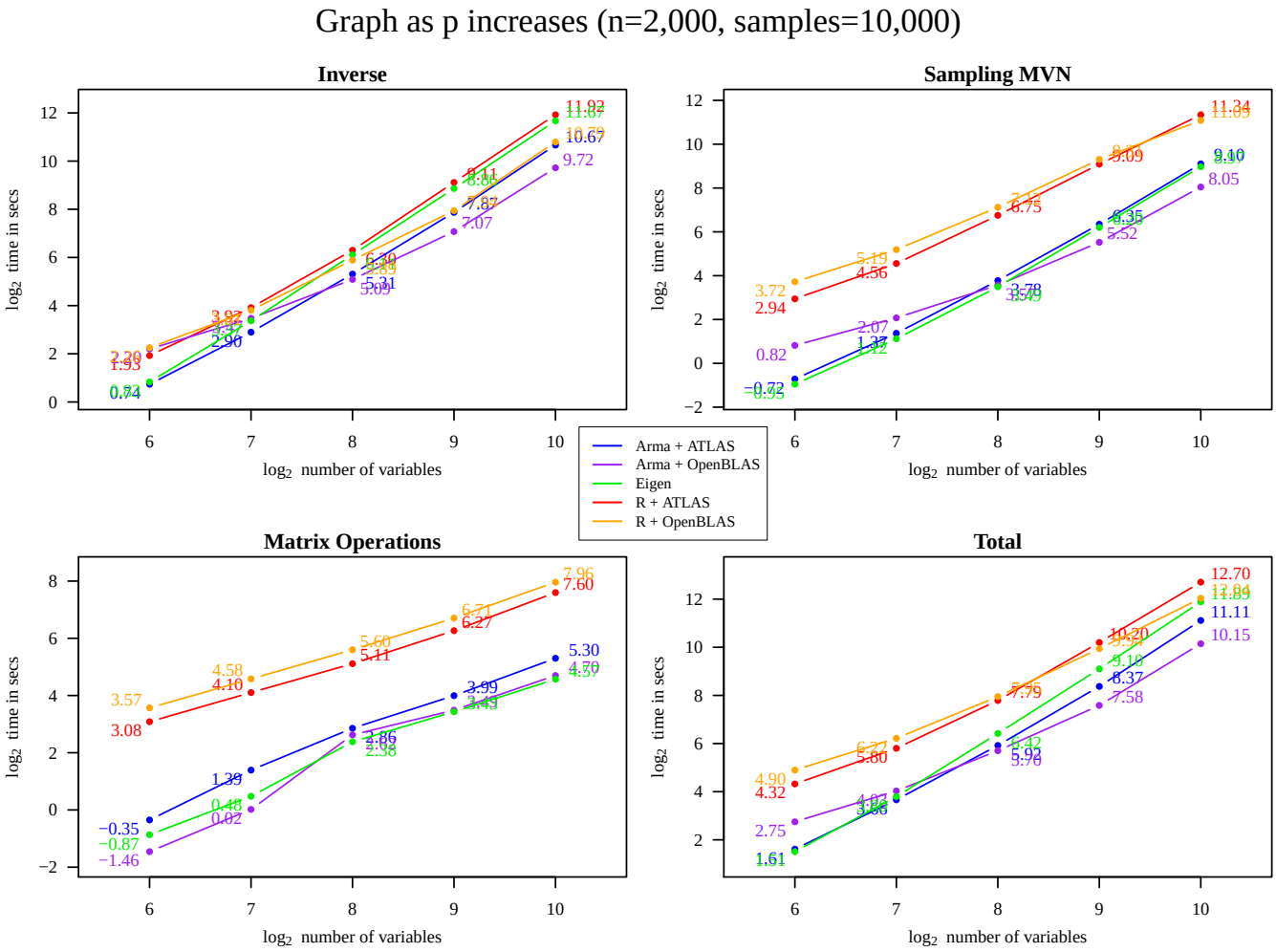
### 3.1 Runtimes as $n$ increases




Observations / impressions regarding the speed comparisons:

- The matrix operations are the dominating cost in performing the Gibbs sampler as $n$ gets comparatively large. The main cost here is incurred by calculating $||\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}^{(s)}||^2$.
- The cost in calculating the matrix inverse and sampling from the multivariate normal distribution mostly is a function of $p$, so we see that the time required for these sections remains mostly constant.
- The R-"only" functions are essentially dominated by the code calling C++-workehorse functions, by a factor of about 5-6 times for the overall running time.
- The parallel processing done for performing the matrix inverse and sampling the multivariate normal distribution is slower for the Armadillo + OpenBLAS than for the sequential processing done for Armadillo + ATLAS or Eigen by about a factor of 1.5 or 2.5 times, respectively. Since these operations are dependent mostly on the size of $p$ which is relatively small here, it seems that the overhead required in the construction and reductions required for parallel processing outweighs the gain incurred by executing instructions concurrently.
- In a comparison of the functions calling sequential C++ code, Armadillo + ATLAS is dominated by Eigen.
- Armadillo + OpenBLAS does comparatively poorly for small sizes of $n$, again presumably due to the overhead incurred for parallel processing. When $n$ gets large Armadillo + OpenBLAS does better than Armadillo + ATLAS, by a factor of about 3.5.
- Eigen is at least tied for the fastest for every size of $n$ observed.

### 3.2 Runtimes as $p$ increases




Observations / impressions regarding the speed comparisons:

- Calculating the matrix inverse and sampling from the multivariate normal distribution are the dominating costs in performing the Gibbs sampler as $p$ gets comparatively large.
- The R-"only" functions are dominated by the code calling C++-workehorse functions that use the Armadillo library, by a factor of about 4 times for the overall running time. However, R + OpenBLAS outperforms Eigen when calculating the matrix inverse for large $p$ by about a factor of 2.
- Armadillo + OpenBLAS does comparatively poorly for small sizes of $p$, again presumably due to the overhead incurred for parallel processing (with the interesting exception of the matrix operations section). However when $p$ gets large Armadillo + OpenBLAS has the best performance, and does better than the second-best competitor, Armadillo + ATLAS, by a factor of about 2.
- The bottleneck for Eigen seems to be the solve method for the Cholesky decomposition. This statement is based on the fact that the cost of the multivariate normal sampling section is about equal for Eigen compared to Armadillo + ATLAS, and which consists of a Cholesky decomposition and matrix multiplication. However, the matrix inverse section for Eigen, which consists of a Cholesky decomposition and a solve method, performs worse than the Armadillo + ATLAS program.

  Perhaps this is due to the fact that this is a general solve method for a Cholesky decomposition - i.e. we solve $\boldsymbol{L}\boldsymbol{L}^T\boldsymbol{X} = \boldsymbol{B}$ for $\boldsymbol{X}$, where we choose $\boldsymbol{B}$ to be the identity matrix and $\boldsymbol{L}$ is the Cholesky decomposition of the matrix of interest. However it would seem that a method that is specialized for calculating the inverse would be far more efficient. Is there a better way to do it in Eigen?