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

A comparison

David Pritchard

July 17, 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
- Practice profiling programs in R and in C++
- Compare the speeds of using various implementations of BLAS and LAPACK

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. If they are run on your computer then the may well tell a different story - and if you are interested enough to read this then it might well be worth finding out!

The specifications of the computing environment for which the results in this document were obtained are shown below. In short, this computer is an Intel i3 64-bit processor with 2 hyper-threaded cores and 4 GB of main memory runing Ubuntu Linux 14.04.

```
sudo lshw -short

## H/W path        Device      Class          Description
## ====================================================
##                             system         Satellite C875 (PSCBAU)
## /0                          bus            PLCSF8
## /0/0                        memory         128KiB BIOS
## /0/4                        processor      Intel(R) Core(TM) i3-3120M CPU @ 2.50GHz
## /0/4/9                      memory         32KiB L1 cache
## /0/4/a                      memory         256KiB L2 cache
## /0/4/b                      memory         3MiB L3 cache
## /0/8                        memory         32KiB L1 cache
## /0/28                       memory         4GiB System Memory
## /0/28/0                     memory         DIMM [empty]
## /0/28/1                     memory         4GiB SODIMM DDR3 Synchronous 1600 MHz (0.6 ns)
## /0/100                      bridge         3rd Gen Core processor DRAM Controller
## /0/100/2                    display        3rd Gen Core processor Graphics Controller
## /0/100/14                   bus            7 Series/C210 Series Chipset Family USB xHCI Host Controller
## /0/100/16                   communication  7 Series/C210 Series Chipset Family MEI Controller #1
## /0/100/1a                   bus            7 Series/C210 Series Chipset Family USB Enhanced Host Controller #2
## /0/100/1b                   multimedia     7 Series/C210 Series Chipset Family High Definition Audio Controller
## /0/100/1c                   bridge         7 Series/C210 Series Chipset Family PCI Express Root Port 1
## /0/100/1c.1                 bridge         7 Series/C210 Series Chipset Family PCI Express Root Port 2
## /0/100/1c.1/0   wlan0       network        RTL8188CE 802.11b/g/n WiFi Adapter
## /0/100/1c.2                 bridge         7 Series/C210 Series Chipset Family PCI Express Root Port 3
## /0/100/1c.2/0   eth0        network        RTL8101/2/6E PCI Express Fast/Gigabit Ethernet controller
## /0/100/1d                   bus            7 Series/C210 Series Chipset Family USB Enhanced Host Controller #1
## /0/100/1f                   bridge         HM76 Express Chipset LPC Controller
## /0/100/1f.2                 storage        7 Series Chipset Family 6-port SATA Controller [AHCI mode]
## /0/100/1f.3                 bus            7 Series/C210 Series Chipset Family SMBus Controller
## /0/1            scsi0       storage
## /0/1/0.0.0      /dev/sda    disk           500GB TOSHIBA MK5075GS
## /0/1/0.0.0/1    /dev/sda1   volume         461GiB EXT4 volume
## /0/1/0.0.0/2    /dev/sda2   volume         3977MiB Extended partition
## /0/1/0.0.0/2/5  /dev/sda5   volume         3977MiB Linux swap / Solaris partition
## /0/2            scsi2       storage
## /0/2/0.0.0      /dev/cdrom  disk           CDDVDW SN-208AB
```

```
## /1                          power          CRB Battery 0
## /2                          power          OEM_Define5
```

```
cat /etc/*release

## DISTRIB_ID=Ubuntu
## DISTRIB_RELEASE=14.04
## DISTRIB_CODENAME=trusty
## DISTRIB_DESCRIPTION="Ubuntu 14.04.4 LTS"
## NAME="Ubuntu"
## VERSION="14.04.4 LTS, Trusty Tahr"
## ID=ubuntu
## ID_LIKE=debian
## PRETTY_NAME="Ubuntu 14.04.4 LTS"
## VERSION_ID="14.04"
## HOME_URL="http://www.ubuntu.com/"
## SUPPORT_URL="http://help.ubuntu.com/"
## BUG_REPORT_URL="http://bugs.launchpad.net/ubuntu/"
```

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

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

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

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

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

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

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

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

2 Progam descriptions

The source code for the following R functions / executable programs is contained in the 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 -only implementation (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 the following:

```
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 More detailed 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 σ 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`

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 (Armadillo provides a median quantile method, but not for arbitrary quantiles). I just emulated the default quantile function from R ; this required about a dozen 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 tasks 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 as far as I can tell, when using the R math library as a standalone library, the random number stream is automatically set up for you, and that setting the seed is accomplished through `set_seed` (declared in `Rmath.h`). However, I have not been able to find any documentation that specifically addresses this point.

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. The easy way to calculate the empirical quantiles of the samples is to sort the data beforehand. As opposed to Armadillo, there was not a method for sorting the rows or columns of a matrix, so this requires just bringing in e.g. `qsort` or `std::sort`.

   As a aside, in order to calculate the emirical quantiles of interest, it is not required to sort the entirety of the data, and there are fast algorithms for obtaining the k-th largest item in a set. However in the interests of (the programmer's) time and simplicity, I just performed a total sort.
3. As far as I could tell, 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. One could presumably construct one in just a few lines - however I did not take the time to do so.

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 -Wall -g3 -O3 -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`

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. 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 they 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.

2.5 Writing `bayes_lm_rcpp_eigen`

I essentially just followed all the steps from section 2.4.2 but using the existing Eigen files - this was trivial to complete after the experiences creating the previous functions / programs.

3 Using the functions / programs

We can use the **`bayes_lm_r`**function as follows.

```
source("Bayes_LM.R")
set.seed(55)
out_time_r <- bayes_lm_r(p = 4, prop_nonzero = 1, print_stats=TRUE)

##
## true beta and posterior quantiles:
## ----------------------------------
##
##     true     2.5%      50%   97.5%
##   0.2403 -0.04785  0.16774  0.3748
##  -3.6248 -3.72480 -3.52636 -3.3242
##   0.3032 -0.14978  0.05437  0.2606
##  -2.2384 -2.70376 -2.46815 -2.2356
##
## true gamma and posterior quantiles:
## -----------------------------------
##
##  true   2.5%    50%  97.5%
##  0.25 0.1772 0.2372 0.3126

out_time_r

##  inverse.elapsed sampling.elapsed  overall.elapsed
##            1.028            3.278            5.770
```

We can use the **`bayes_lm_arma`**and **`bayes_lm_eigen`** programs through the usual command-line syntax.

```
./bayes_lm_arma -p=3 -prop_nonzero=1 -decomp_method=eigen -seed=93  \
      -write_samples=true -samples_file_loc=Samples_Test_Arma.dat   \
      -write_ctime=true -ctime_file_loc=CTime_Test_Arma.dat
cat Samples_Test_Arma.dat | head -n5
cat CTime_Test_Arma.dat

./bayes_lm_eigen -nsamp=11089 -n=52 -p=8 -prop_nonzero=0.5 -seed=26  \
      -print_stats=true



## -0.19323 -1.17144 2.22477 0.25
## -0.101888 -1.20174 2.07333 0.230466
## -0.0920114 -1.12371 1.97472 0.20906
## -0.157833 -1.07462 2.05615 0.229421
## -0.0398933 -1.12509 2.06389 0.275206
## 0.111305 0.142743 0.358414
##
## Parameter specifications:
## -------------------------
## n:  52
## p:  8
## prop_nonzero:  0.5
## true_beta_sd:  2
## true_sigma:  2
## sd_x:  2
## nsamp:  11089
## print_stats:  1
## write_ctime:  0
## ctime_file_loc:  Comp_Time_Eigen.dat
## write_samples:  0
## samples_file_loc:  Samples_Eigen.dat
## decomp_method:  c
## seed:  26
##
## Elapsed time:
## -------------
## Inverse:          0.0290
## Sampling normal:  0.0197
## Overall:          0.0568
##
##  true beta       2.5%        50%      97.5%
## ------------------------------------------
##     0.0000     0.1070     0.3646     0.6352
##     0.8614     0.4281     0.7553     1.0944
##     0.0000    -0.3580    -0.1263     0.1008
##    -0.4482    -0.4063    -0.1532     0.1090
##     2.3183     2.1320     2.4436     2.7528
##     0.0000    -0.1635     0.0776     0.3112
##    -1.1638    -1.4465    -1.1445    -0.8360
##     0.0000    -0.3386    -0.0791     0.1781
##
## true gam 2.5%        50%      97.5%
## ----------------------------------
##   0.2500   0.2141   0.3334   0.4924
```

The syntax for using the R functions created via Rcpp is the same as the R -only version.

```
source("Bayes_LM_Rcpp_Arma.R")
source("Bayes_LM_Rcpp_Eigen.R")
set.seed(22)

# Sampler examples with arbitrary parameter specifications
out_time_rcpp_arma <- bayes_lm_rcpp_arma(n=125, p=20, nsamp=1.5e4, decomp_method="eigen")
out_time_rcpp_eigen <- bayes_lm_rcpp_eigen(n=500, p=10, true_beta_sd=3, true_sigma=4, sd_x=5)

out_time_rcpp_arma

##  inverse sampling  overall
## 1.319923 4.070896 5.464781

out_time_rcpp_eigen

##    inverse   sampling    overall
## 0.05075003 0.03449665 0.13406547
```

4 Profiling the functions / programs

4.1 Profiling `bayes_lm_r`

4.1.1 The cost of using `system.time`

Lets take a look at the time that it takes to execute 10,000 samples for a data set with 100 observations and 15 variables. First we make an observation about the cost of recording the elapsed time for each of the sections. Recording the elapsed time in lmr is done via `system.time` which in turn calls `proc_time` twice. `system.time` is called twice during each scan of the sampler, one for the matrix inversion, and once for the multivariate normal sampling step.

```
set.seed(11)

# Time the individual sections by default
(time_r <- bayes_lm_r(n=100, p=15, nsamp=1e4))

##  inverse.elapsed sampling.elapsed  overall.elapsed
##            1.247            4.167            6.922

# The function returns 0 for the sections when they are not timed
(time_r_notime <- bayes_lm_r(n=100, p=15, nsamp=1e4, time_sections=FALSE))

##         inverse        sampling overall.elapsed
##           0.000           0.000           5.593
```

So in this instance we see that calling `system.time` twice during each scan makes the sampler run 24% longer! This is a tremendous cost and has major implications for profiling R code.

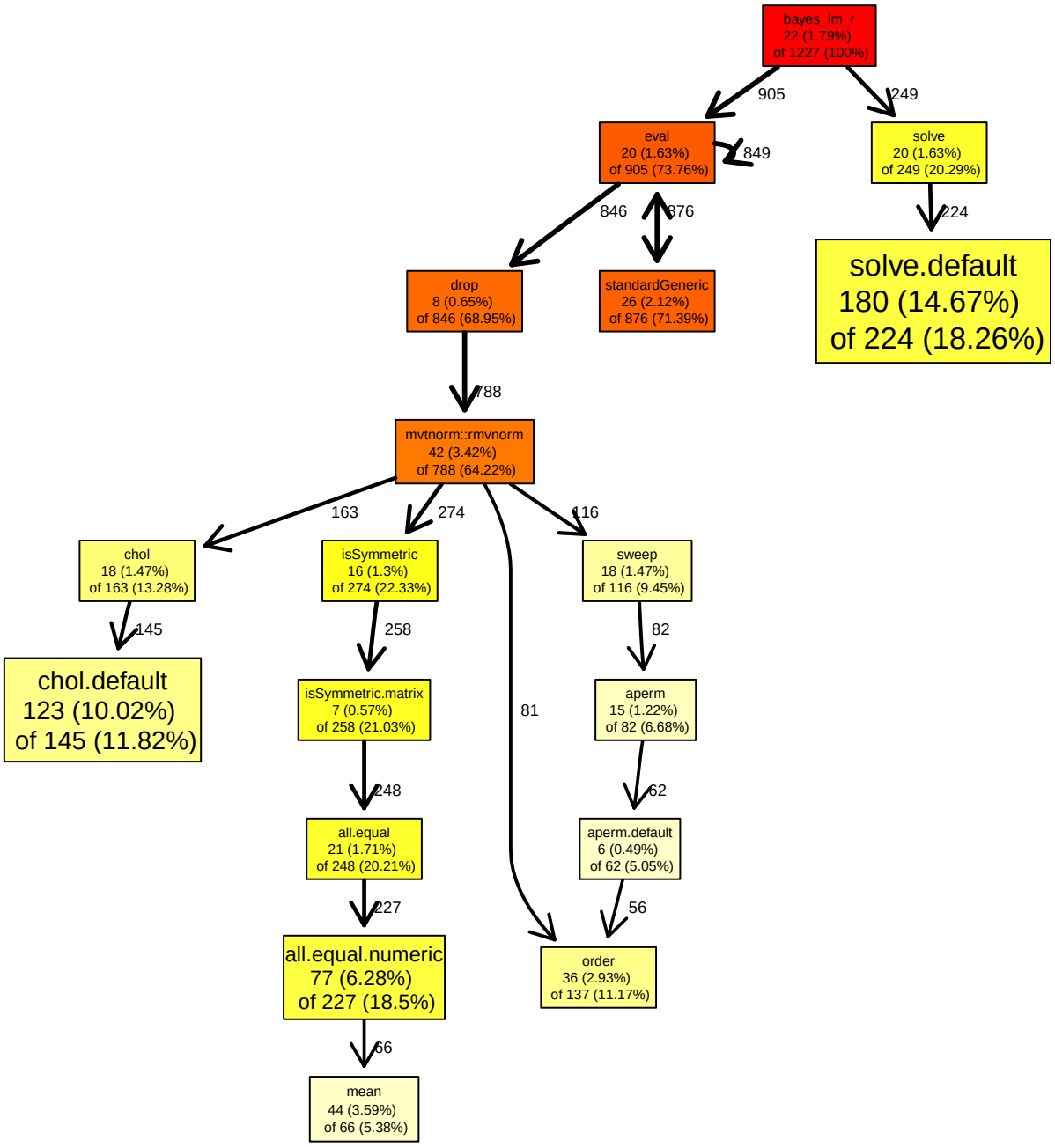
Now these figures seem to suggest that the cost of the matrix inversion and the multivariate normal sampling step was 78% of the elapsed time of the sampler. However, it seems that the cost of calling `system.time` may be included in the elapsed time values for these steps - since the sum time of the two steps is larger than the entire algorithm time when we didn't record the elapsed time for these steps. So we have another reason to be careful when using `proc_time` to profile R code.

4.1.2 Profiling `bayes_lm_r`with `proftools`

```r
# Profile an arbitrary instance of bayes_lm_r
set.seed(82)
library(proftools)
pd <- profileExpr( bayes_lm_r(n=100, p=15, nsamp=1e4, time_sections=FALSE) )

# Print the 10 function calls that bayes_lm_r spent the most time executing
head(prof_stats <- funSummary(pd, FALSE), 10)
```

```
##                   total.pct gc.pct self.pct gcself.pct
## solve.default         18.26   0.24    14.67       0.00
## chol.default          11.82   0.00    10.02       0.00
## all.equal.numeric     18.50   1.14     6.93       0.65
## mean                   5.38   0.16     3.75       0.16
## mvtnorm::rmvnorm      64.22   1.55     3.50       0.08
## order                 11.17   0.00     2.93       0.00
## %*%                    2.53   0.00     2.53       0.00
## vapply                 3.67   0.00     2.36       0.00
## standardGeneric       71.39   1.55     2.12       0.00
## as.matrix              2.77   0.00     2.04       0.00
```

```r
# Plot a call graph of the exectution instance
plotProfileCallGraph(pd, total.pct=5)
```



So the profiler estimates that about 25% of the running time of the program was spent on the calls to `solve.default` and `chol.default`. It is interesting to note that `all.equal.numeric` and mean `mean` required so much time to perform during the sampler, since neither of them is explicitely called during `bayes_lm_r`.

4.2 Profiling `bayes_lm_arma`

4.2.1 The cost of using `clock_gettime`

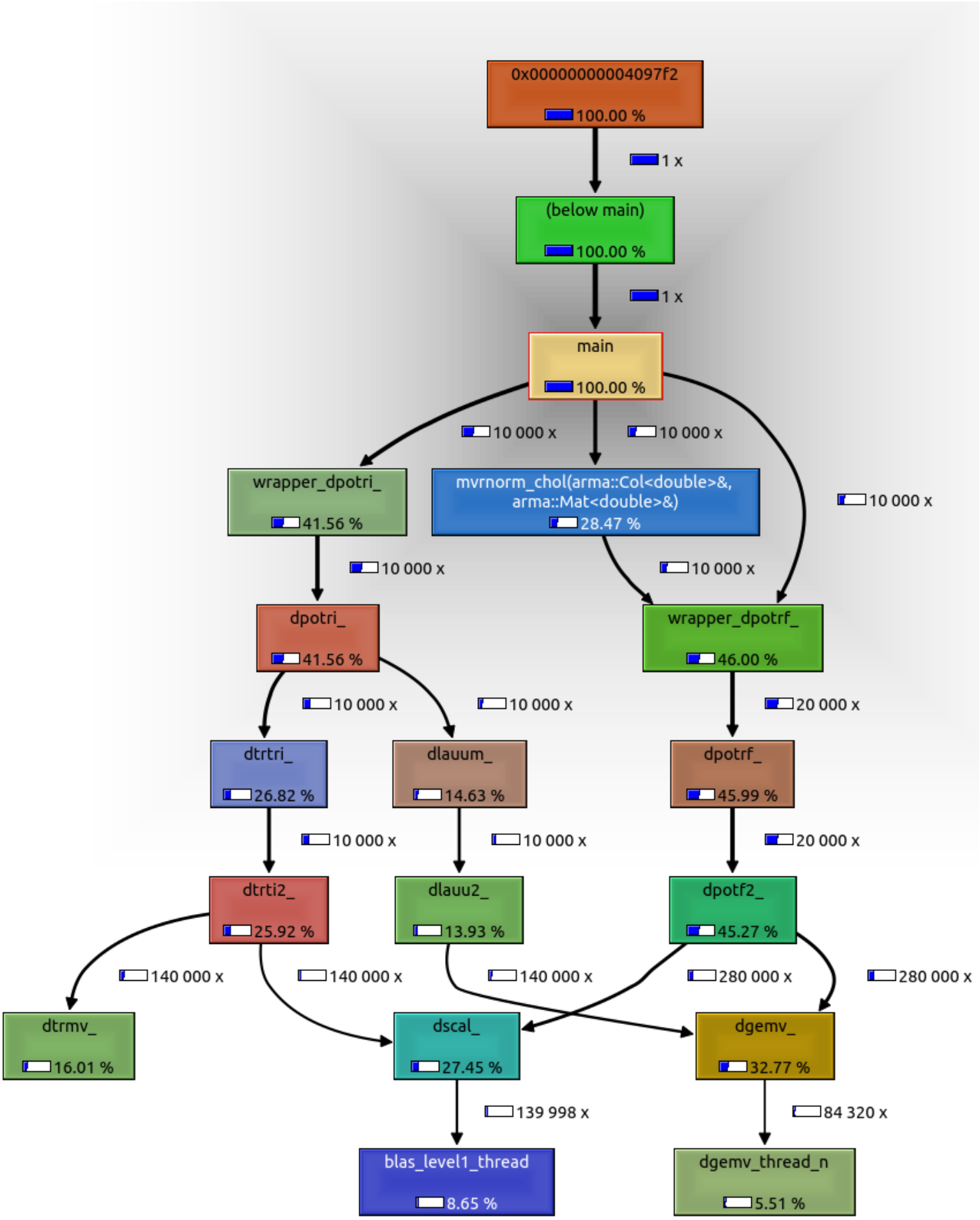
We can compile a program that does not track the elapsed time of the matrix inversion and the multivariate normal sampling step by declaring the `NO_TIMER` macro. Then a comparison for of running times of the samplers is shown below.

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

time ./bayes_lm_arma_no_timer -n=100 -p=15 -nsamp=10000

time ./bayes_lm_arma -n=100 -p=15 -nsamp=10000

##
## real 0m0.905s
## user 0m1.148s
## sys 0m2.432s
##
## real 0m1.045s
## user 0m1.304s
## sys 0m2.744s
```

So for this particular instance there was a small cost in invoking the system timer. However this cost is far cheaper than when using the R system timer.

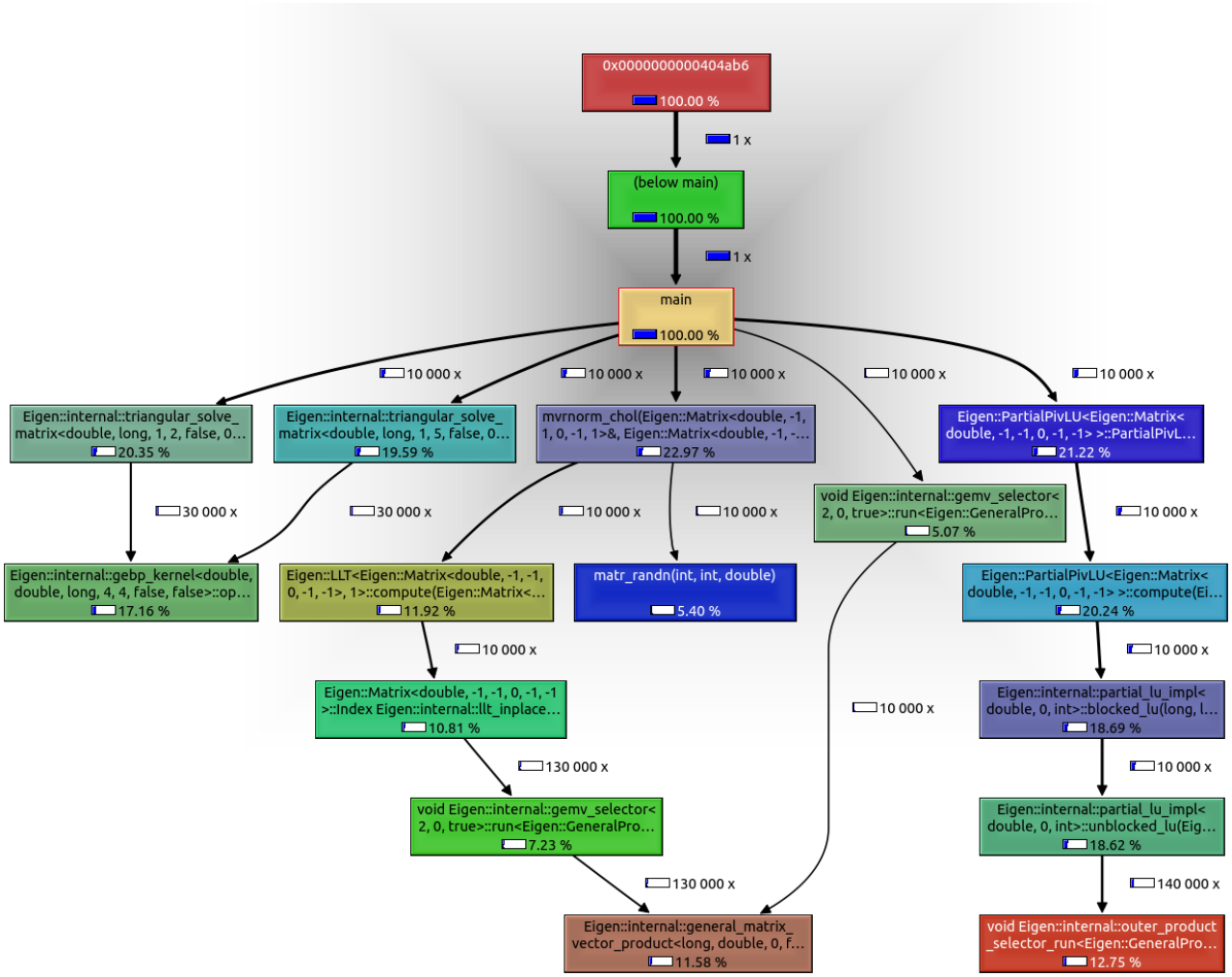
4.2.2 Profiling `bayes_lm_arma`with `callgrind`

The call graph for an instance of **`bayes_lm_arma`**execution as generated by `callgrind` is shown below. The call `dpotri` computes the inverse of a real symmetric positive definite matrix using the Cholesky factorization. The call `dpotrf` computes the Cholesky factorization of a real symmetric positive definite matrix.



4.2.3 Profiling `bayes_lm_eigen` with `callgrind`

The call graph for an instance of **`bayes_lm_arma`**execution as generated by `callgrind` is shown below. The call `dpotri` computes the inverse of a real symmetric positive definite matrix using the Cholesky factorization. The call `dpotrf` computes the Cholesky factorization of a real symmetric positive definite matrix.



4.2.4 Profiling `bayes_lm_rcpp_arma`and `bayes_lm_rcpp_eigen`

TODO: it would be a good exercise to profile the Rcpp versions of Bayesian linear model programs, since presumably we should get comparable results to when we used the standalone C++ versions. But I have not yet learned how to do this.

5 Comparison of program speeds

Elapsed time as n increases