Acceleration of Metropolis-Hasting Algorithm

Tairan

May 3, 2018

Movitation

Bayesian Framewrok

Let's begin with the brief introduction to the Bayesian framework. In Bayesian statistics, we would like to figure out what the parameter space will look like after considering the available data. From the mathematics point of view, we can summarize the Bayesian framework as:

$$f(\theta|X) = \frac{f(X|\theta) \times f(\theta)}{f(X)}$$

The notation can be explained as following:

- $f(\theta)$, or the prior, is our initial belief about the distribution of the parameter.
- $f(X|\theta$, or the likelihood, is the distribution of the data given the parameters.
- $f(\theta|X)$, or the posterior, is the distribution of the parameter given the data

Since the denominator $f(X)$ doesn't contain the parameter θ, the equation can be simplified as:

$$Posterior \propto Likelihood \times Prior$$

However, the posterior distribution sometimes is hard to solve in mathematics form. Then, we need to introduce the sampling algorithm to explore the its equilibrium distribution.

Sampling Algorithm

The Markov chain Monte Carlo (MCMC) methods includes a class of algorithm for sampling from a probability distribution. We construct the Markov chain to depict the desired distribution, and then study its properties and conduct the statistics test. The most common applied sampling algorithms in MCMC are Gibbs sampling, Metropolis-Hasting sampling, and slice sampling. (Wikipedia contributors 2018b)

- Gibbs sampling
  - When to apply:
    Gibbs sampling works well when the conditional distribution for each random variable is in a analytic form, like belonging to a known distribution. In this case, we can directly generate the sample distribution from this known distribution.
  - Strength:
    If Gibbs sampling is applicable, it is definitely the most efficient sampling algorithm.
  - Weakness:
    The requirement for the conditional distribution is extreme higher than other sampling algorithm.
- Metropolis-Hasting sampling
  - When to apply:
    Metropolis-Hasting sampling can work in most of cases.

  - Strength:
    You don't need to worry too much about the mathematics deductions for the conditional distribution.
  - Weakness:
    Building a well-tuned proposal distribution demands experience and might take long time. If the step size is too small random walk causes slow decorrelation. If the step size is too large there is great inefficiency due to a high rejection rate.
- Slice sampling
  - When to apply:
    Slice sampling performs well in unimodal distributions and some steep multi-modals distributions.
  - Strength:
    Compared to Metropolis-Hasting sampling, there is no need to manually tune the proposal function or proposal standard deviation.
  - Weakness:
    When it comes to multidimensional distributions, we can sample each random variable using one-dimensional slice sampling, in a manner similar to Gibbs sampling. However, this approach severely suffer from complication introduced by dimensionality.

As we can see, Metropolis-Hasting is the most widely applicable sampling algorithm. However, the evaluation of the posterior and the proposal distribution is a significant run time bottleneck. Then we are seeking a method that can reduce the run time of Metropolis-Hasting.

Introduction of C++ to sampling algorithm

C++'s advantage in computation efficiency

C++ is a general-purpose programming language. It has imperative, object-oriented and generic programming features, while also providing facilities for low-level memory manipulation.

The reason why C++ is fast in computation can be summarized as following:

- Compact Data Structure:
  C++ is intrinsically stingy with memory – a C++ struct has no memory overhead if there are no virtual functions. Smaller things run faster due to caching, and are also more scalable.
- Template Meta-programming:
  Huge gains to be made with the kind of evaluation you can implement with expression templates and the kind of memory and indirection savings you achieve by avoiding virtual functions with the curiously recurring template pattern.

Therefore, the ideal procedure to analyze the posterior distribution is to sample in C++ for computation efficiency and study in R for mass statistics analysis packages.

RCpp

The RCpp package provides R functions as well as C++ classes which offer a seamless integration of R and C++. Many R data types and objects can be mapped back and forth to C++ equivalents which facilitates both writing of new code as well as easier integration of third-party libraries. (Eddelbuettel et al. 2011)

The core packages in RCpp includes: (Wikipedia contributors 2018a)

- Rcpp
  - Rcpp provides a powerful API on top of R, permitting direct interchange of rich R objects between R and C++
  - Rcpp sugar gives syntactic sugar such as vectorised C++ expression

  - Rcpp modules provide easy extensibility using declarations and Rcpp attributes greatly facilitates code integration. (RCpp 2018)
- RcppArmadillo
  - Rcpp connects R with the powerful Armadillo templated C++ library for linear algebra
  - Armadillo aims towards a good balance between speed and ease of use, and its syntax is deliberately similar to Matlab which makes it easy to port existing code
- RcppEigen
  - RcppEigen gives R access to the high-performance Eigen linear algebra library
  - Eigen is templated and highly optimized, which provides a wide variety of matrix methods, various decomposition and includes support for sparse matrices.
- RInside
  - The RInside package provides C++ classes that make it easier to embed R in C++ code by wrapping the existing R embedding API in an easy-to-use C++ class
  - Over a dozen basic example are included in the package, as well as more specialized example showing use of RInside with MPI for parallel computing, Qt for cross-platform GUI, Wt for web applications, as well as the Armadillo and Eigen templated C++ libraries for linear algebras

Stan

Stan is a probabilistic programming language for statistical inference written in C++. The Stan language is used to specify a (Bayesian) statistical model with an imperative program calculating the log probability density function.

Stan implements gradient-based Markov chain Monte Carlo (MCMC) algorithms for Bayesian inference, stochastic, gradient-based variation Bayesian methods for approximate Bayesian inference, and gradient-based optimization for penalized maximum likelihood estimation.

The package `RStan` in R can be employed to connect R and Stan. Therefore, the sample distribution is simulated in Stan and then transport back to R.

An Example: Logistic Model

Artificial Data

The artificial data is generated as following:

- The sample size n is 50
- X_1: continuous variable, subject to a normal distribution with mean 0 and variance 1
- X_2: continuous variable, subject to a normal distribution with mean 0.5 and variance 1
- X_3: continuous variable based on the absolute value of X_1
- Y: binary outcome, generated based on:

$$logit(E(Y)) = 1 + 3 \times X_1 + 3 \times X_2 - 3 \times X_3$$

Bayesian Modeling

Based on the artificial data, we apply the logistic regression model:

$$\begin{aligned} logit(E(Y)) &= logit(p_i) \\ &= \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_3 \end{aligned}$$

Also provide the parameter with noninformative prior:

$$\vec{\beta} \sim N_4\left(\begin{bmatrix}0\\0\\0\\0\end{bmatrix}, \begin{bmatrix}100 & 0 & 0 & 0\\ 0 & 100 & 0 & 0\\ 0 & 0 & 100 & 0\\ 0 & 0 & 0 & 100\end{bmatrix}\right)$$

By easy mathematics deduction, we can find out the posterior distribution as:

$$\log f(\vec{\beta}|X,Y) \propto \sum\{y_i \times \log(p_i) + (1-y_i)\times\log(1-p_i)\} - \frac{\vec{\beta}^\intercal\vec{\beta}}{2\times 10}$$

$$logit(p_i) = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_3$$

Since the posterior distribution is not subject to any known distribution, we need to employ the Metropolis-Hasting sampling algorithm.

Sampling Algorithm

Generally speaking, our sampling algorithm consists of the following steps:

1. Set a starting point to β
2. Sample the proposal distribution, which is centered and symmetric around $\beta^{(s)}$

$$\beta^* \sim N(\beta^{(s)}, \sigma_p^2)$$

3. Compute the acceptance ratio:

$$r = \frac{f(\beta^*|X,Y)}{f(\beta^{(s)}|X,Y)} * \frac{N(\beta^{(s)}|\beta^*)}{N(\beta^*|\beta^{(s)})}$$

4. Update the sequence:

$$\beta^{(s+1)} = \beta^* \text{ if } r > u \sim U(0,1)$$
$$\beta^{(s+1)} = \beta^{(s)} \text{ if } r < u \sim U(0,1)$$

5. Repeat Step2 to Step4 until a converged chain achieved

Computation Method

Let's see the sample code in Cpp, Stan, and R. Their run time will be compared later. (Stable Markets 2018)

1. Complete all sampling in R, denoted as R:

We define a function to compute the logarithm of posterior in R at very beginning.

```r
# compute the log posterior of beta vector
log_posterior<-function(beta, X, Y){

  # calculate likelihood
  p_i <- invlogit(X %*% beta)
  likelihood <- sum(dbern(Y, p_i, log = T))

  # calculate prior
  prior <- dmvn(x = beta, mu = rep(0,p), Sigma = sqrt((10^2)*diag(p)), log = T)

  # calculate the posterior
  log_post <- likelihood + prior

  return(log_post)
}
```

Then a for loop is employed to generate the sample distribution with `for` loop.

```
# Generate the sample distribution

for(i in 2:iteration){
  beta_s <- BETA[i-1, ]

  # sample from proposal distribution
  beta_star <- mvrnorm(n = 1, beta_s, Sigma = sqrt(jump_v*diag(p)))

  # calculate acceptance probability
  r_num <- log_posterior(beta_star, X, Y ) +
           dmvnorm(x = beta_s, mean = beta_star, sigma = sqrt(jump_v*diag(p)))
  r_denom <- log_posterior(beta_s, X, Y ) +
           dmvnorm(x = beta_star, mean = beta_s, sigma = sqrt(jump_v*diag(p)))

  r <- r_num - r_denom
  rmin<-min(r,log(1))

  # accept or reject proposal
  if( rmin > log(runif(n = 1, min = 0, max = 1)) ){
    BETA[i, ] <- beta_star
  }else{
    BETA[i, ] <- beta_s
  }
  accept[i] <- rmin
}
```

2. Compute the posterior in C++ and simulate the sample distribution in R, denoted as RCpp:

We define a function to compute the logarithm of posterior in C++ at very beginning.

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

vec invlogit(vec x){
  vec y = exp(x)/(1 + exp(x));
  return(y);
}
```

```cpp
// [[Rcpp::export]]
double log_post(vec b, vec Y, mat X, vec b_s, double jump_v) {
  int n = X.n_rows;
  int p = X.n_cols;
  vec xb;
  vec pi;
  vec mu;
  mat sd_prior(p, p);
  mat sd(p, p);
  double likelihood = 0;
  double prior = 0;
  double proposal;
  double log_posterior = 0;

  // linear predictor
  xb =  X*b;

  // correct for numerical issues
  for( int i=0; i<n; i++){
    if( xb[i] > 10 ){
      xb[i] = 10;
    }else if( xb[i] < -10 ){
      xb[i] = -10;
    }else{
      xb[i] = xb[i];
    }
  }

  // apply inverse link function
  pi = invlogit(xb);

  // compute log likelihood
  for(int i=0; i<n; i++){
    likelihood = likelihood + R::dbinom(Y[i], 1, pi[i], 1);
  }

  // compute log prior contribution for each parameter
  for (int i = 0; i < p; i++)
  {
      prior = prior + R::dnorm4(b[i], 0, 10, 1);
  }

  // compute the proposal density
  for (int i = 0; i < p; i++)
  {
      proposal = proposal + R::dnorm4(b_s[i], b[i], sqrt(jump_v), 1);
  }

  // evaluate log posterior as sum of likelihood and prior
  log_posterior = likelihood + prior;
  return log_posterior;
}
```

Then by using the sourceCpp function, we can import this C++ function in R. Then generate the sample distribution by `for` loop in R.

3. Sampling in C++, denoted as C++

Since the `for` loop in R is somehow a bottleneck for computation efficiency, we would like to complete all the sampling task in C++. So we define a function in C++ to update the sequence of posterior, which is also involved the `for` in C++.

```cpp
// [[Rcpp::export]]
List for_log_post(mat X, vec Y, int iteration, double jump_v){
    // create shell
    int p = X.n_cols;
    mat BETA(iteration, p);
    vec accept_rate(iteration);

    // starting value
    BETA.row(0) = ones<vec>(p).t();
    vec beta_0(p);
    mat sd(p, p);
    vec beta_c(p);
    double r_denum = 0;
    double r_num = 0;
    double r = 0;
    double rmin = 0;
    double uni = 0;

    for(int i = 1; i < iteration; i++){
        beta_0 = trans(BETA.row(i-1));

        sd = diagmat(jump_v * ones<vec>(p));

        // sample from proposal distribution
        beta_c = mvnrnd(beta_0, sd);

        // ratio computation
        r_num = log_post(beta_c, Y, X, beta_0, jump_v);

        r_denum = log_post(beta_0, Y, X, beta_c, jump_v);

        // acceptance ratio
        r = exp(r_num - r_denum);
        rmin = fmin(r, 1);

        //decision
        uni = randu<double>();
        if (rmin > uni){
            BETA.row(i) = beta_c.t();
        } else {
            BETA.row(i) = beta_0.t();
        }
        accept_rate(i) = rmin;
    }
    List result;
    result["BETA"] = BETA;
    result["accept_rate"] = accept_rate;
    return result;
}
```

4. RStan We define the Bayesian model in Stan as following:

```
data {
  int N;
  int p;
  matrix[N, p] x;
  int y[N];
}

parameters {
  vector[p] beta;
}

model{
  // prior
  beta ~ normal(0, 100);

  //likelihood
  y ~ bernoulli_logit(x * beta);
}
```

Then we can implement this model by commanding in R:

```
res_rstan <- stan(file = 'posterior_in_stan.stan',
                  data = list(x = X, y = Y, N = nrow(X), p = ncol(Y)),
                  iter = iteration, chains = 1)
```

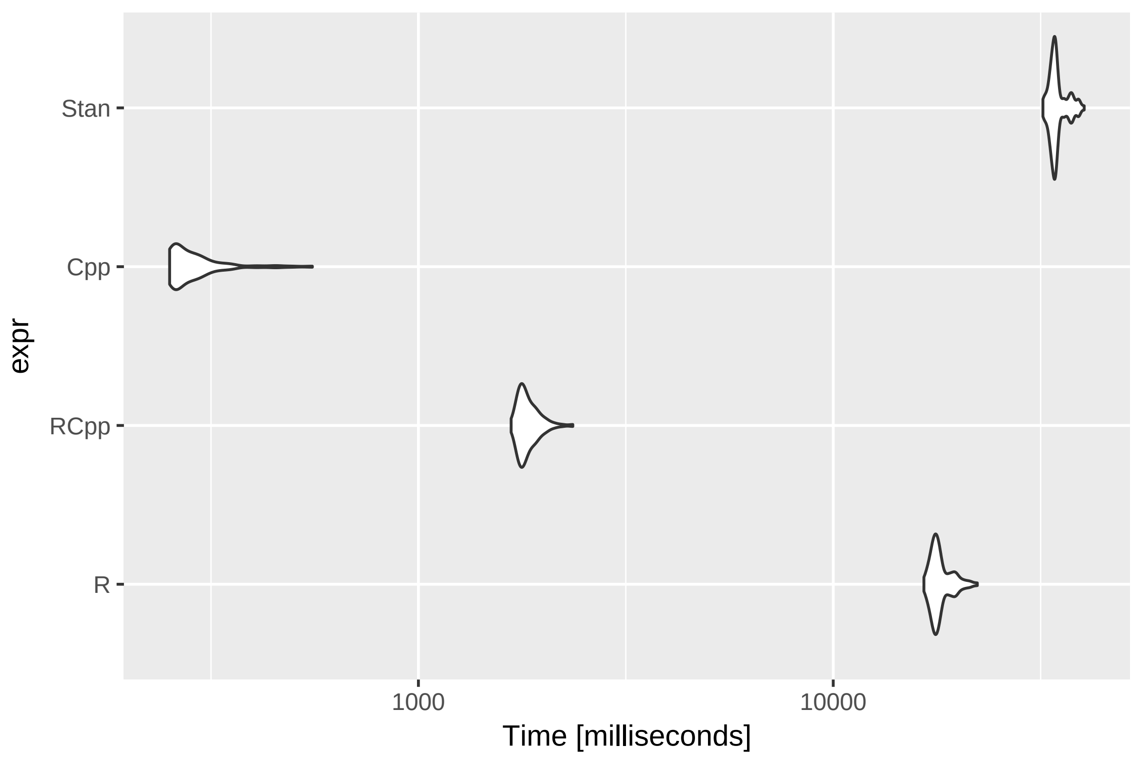
Run Time Comparison

Run Time by Method

All these approaches will generate similar posterior distribution. I attach the trace plots for these different as appendix. Now let's focus on the run time of these different approaches. The R function `microbenchmark` is used here to measure the running time accurately, which is a good substitution for the `system.time(replicate(repeat_time, function))` expression. The result is as following:

Table 1: Run Time Summary with Sample Size 50

method	min	lq	mean	median	uq	max	neval
Pure R	16541.4761	17448.0089	18159.5563	17805.9182	18646.854	22250.6081	100
RCpp	1673.0167	1757.6993	1840.5268	1804.0076	1899.491	2356.4302	100
Cpp	251.3411	258.2004	290.7577	273.3687	299.332	555.1278	100
Stan	32026.7448	33644.3007	34746.3294	34243.7151	35054.643	40277.1960	100



Conclusion

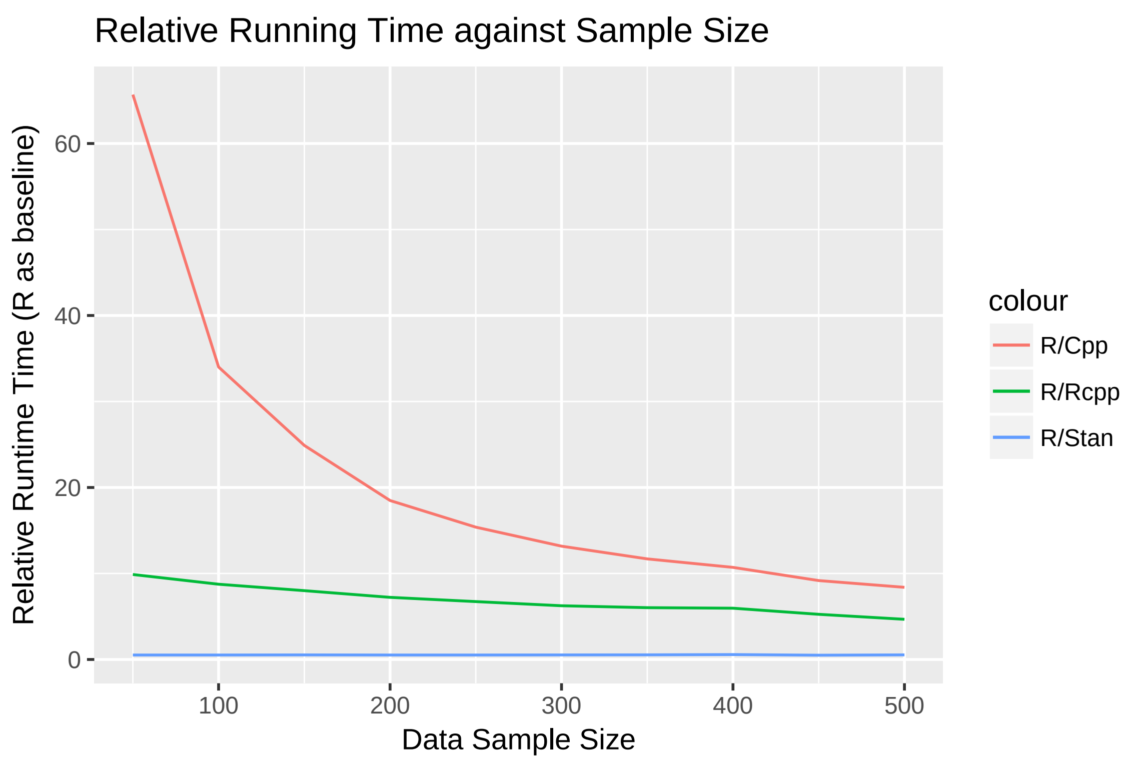
As we can see, if we generate the sample distribution in C++, the run time can be much shorter than any other method. Even when we only put the computation of posterior in C++, it can still save around 90% run time compared to the pure R computation. However, due to time cost in communication between R and Stan, RStan returns the worst run times. If we also consider the hours spent on coding and mathematics deduction, RStan will be the most efficient one.

Run Time by Sample Size

In previous case, we study the case with sample size 50. We are also interested in the relationship between the run time and the sample size, which is from 50 to 500 with increment of 50. We set the run time of R as the baseline.

Table 2: Relative Run Time Summary against Sample Size

Sample Size	R/Rcpp	R/Cpp	R/Stan
50	9.882080	65.689113	0.5193266
100	8.749257	34.008443	0.5206681
150	8.003832	24.891335	0.5331680
200	7.225733	18.484568	0.5202131
250	6.730718	15.387929	0.5200740
300	6.248238	13.175251	0.5285307
350	6.023071	11.696660	0.5369588
400	5.961274	10.709417	0.5688523
450	5.253168	9.174063	0.4965358
500	4.673372	8.391021	0.5362079

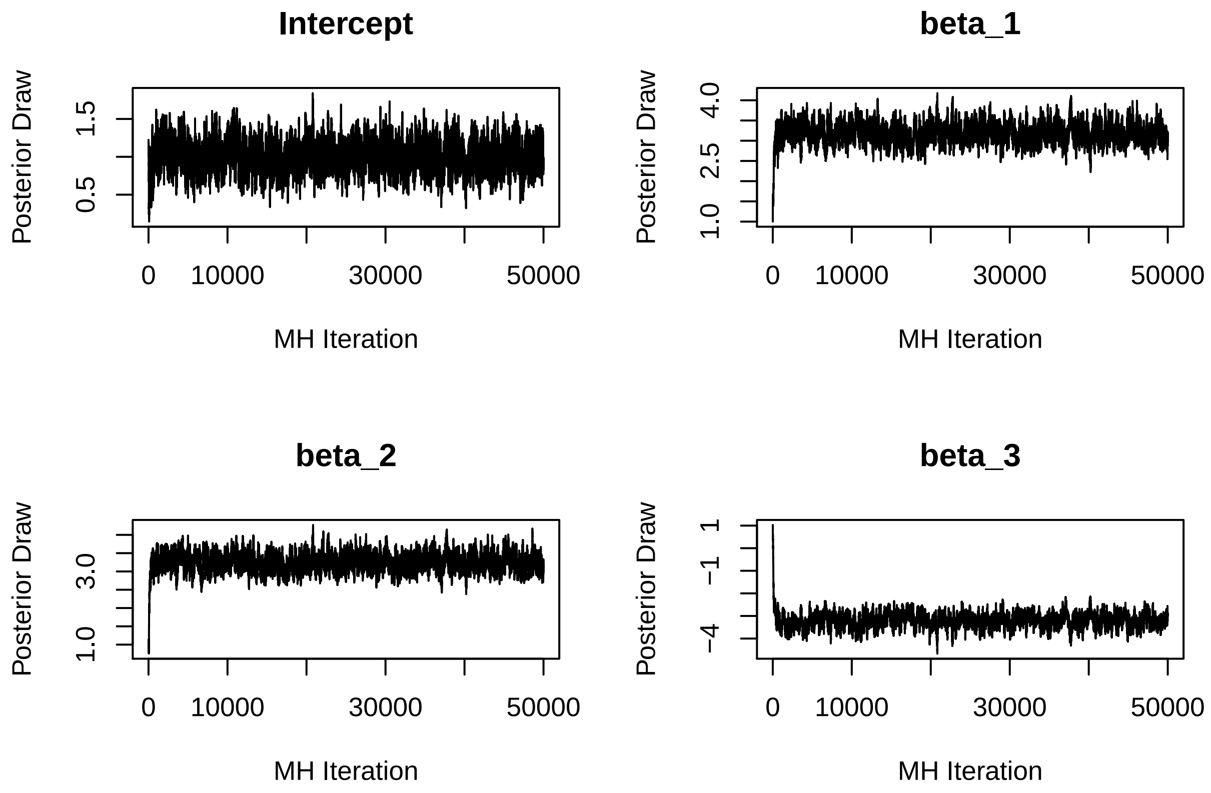


Conclusion

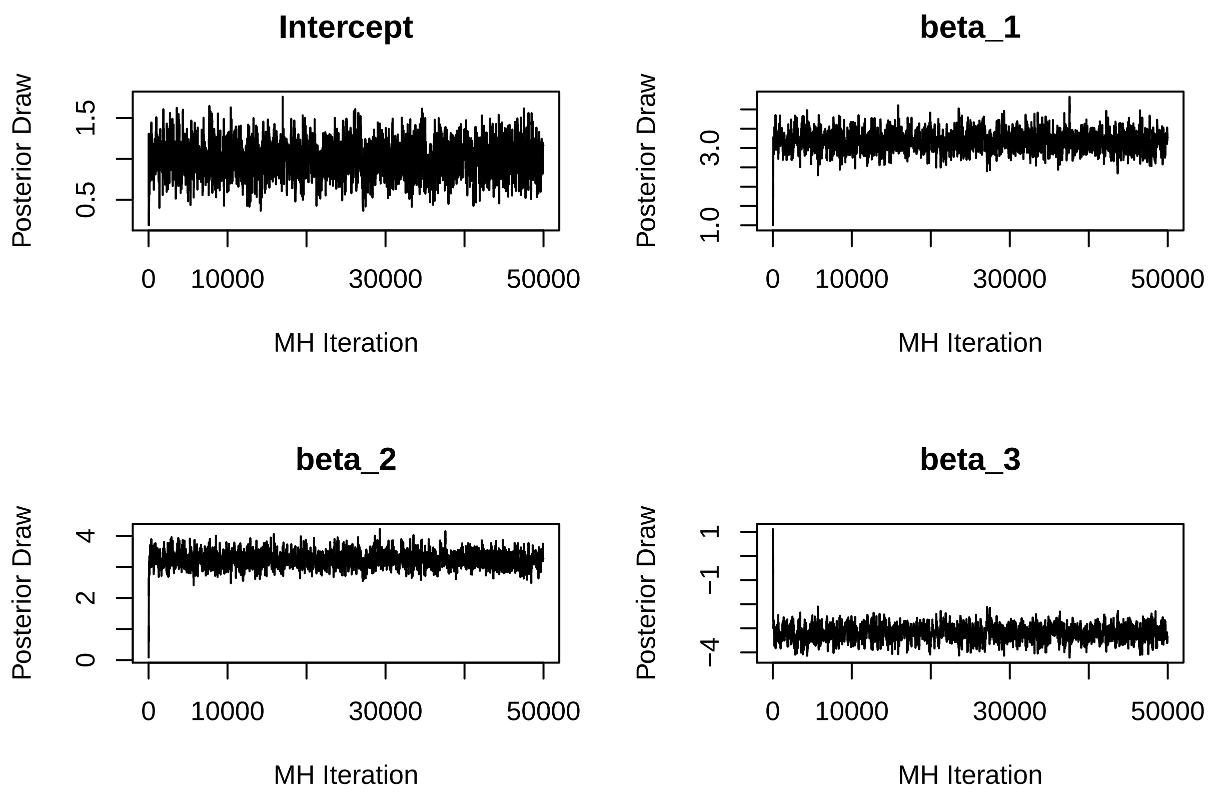
As we can see, C++ method is always dominating. As the sample size increase, all these run time ratios are dropping. And an interesting result is from Stan. Although built on C++, Stan is even slower than R method. We believe the tuning procedure in Stan costs significantly in the 'warm-up' procedure, which will automatically find out the best proposal distribution. Therefore, if you want to speed up your sampling, you can try to sample in C++.

Appendix: Trace Plot for Different Methods

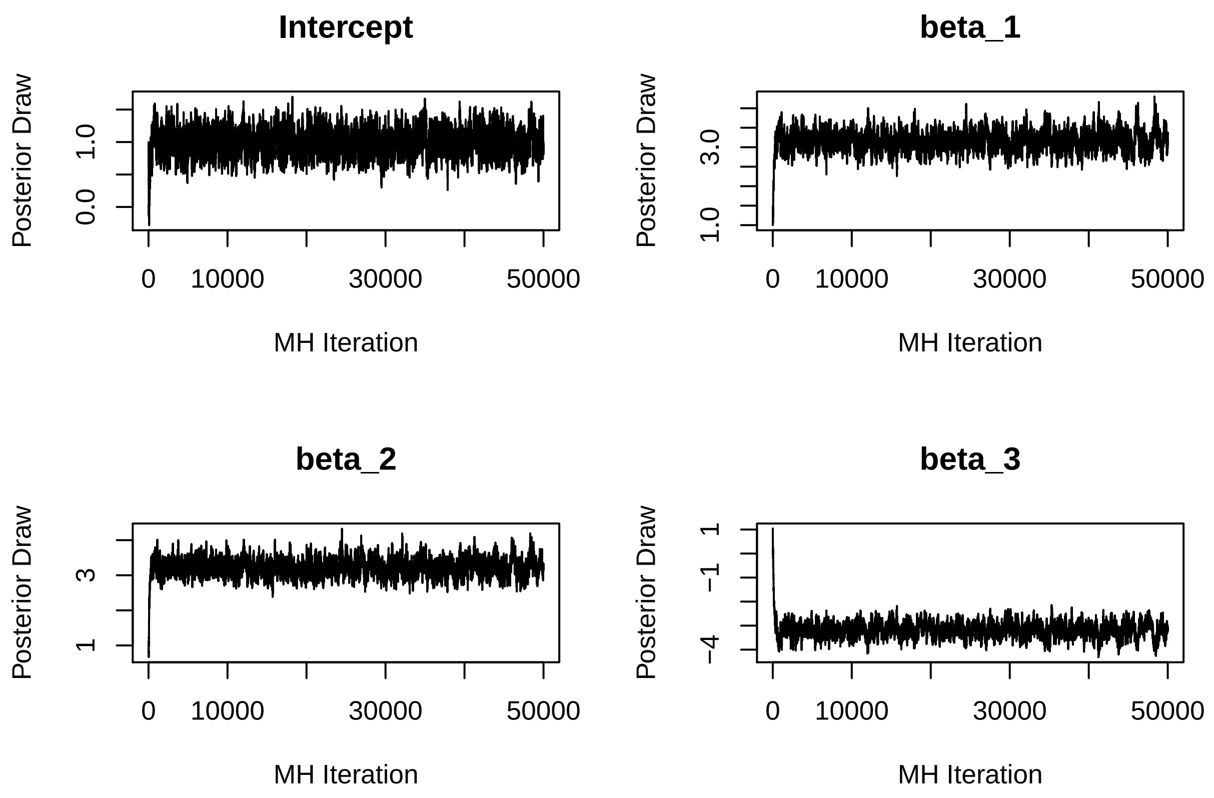
1. R



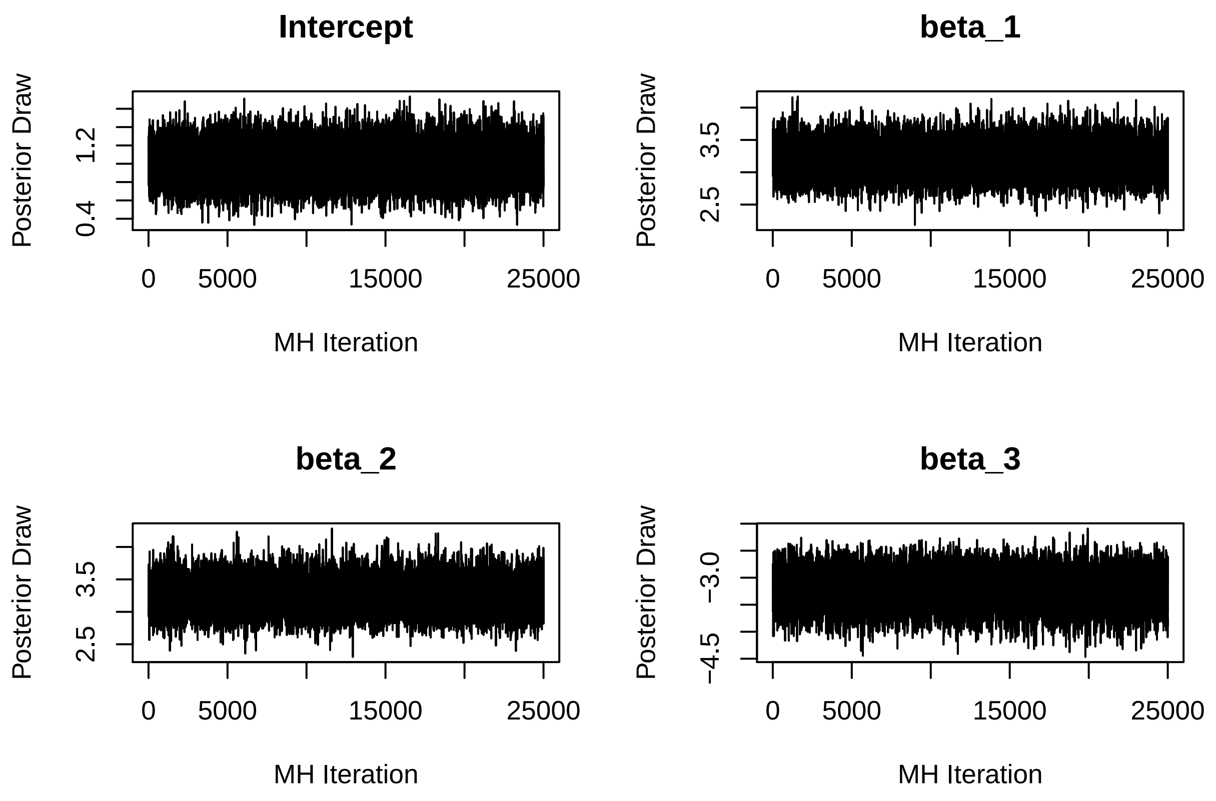
2. RCpp


Intercept
beta_1
beta_2
beta_3
Posterior Draw
MH Iteration
0
10000
30000
50000
0.5
1.5
1.0
3.0
0
2
4
−4
−1
1

3. Cpp


Intercept
Posterior Draw
0.0
1.0
0
10000
30000
50000
MH Iteration
beta_1
Posterior Draw
1.0
3.0
0
10000
30000
50000
MH Iteration
beta_2
Posterior Draw
1
3
0
10000
30000
50000
MH Iteration
beta_3
Posterior Draw
−4
−1
1
0
10000
30000
50000
MH Iteration

4. RStan


Intercept
Posterior Draw
0.4
1.2
0
5000
15000
25000
MH Iteration
beta_1
Posterior Draw
2.5
3.5
0
5000
15000
25000
MH Iteration
beta_2
Posterior Draw
2.5
3.5
0
5000
15000
25000
MH Iteration
beta_3
Posterior Draw
−4.5
−3.0
0
5000
15000
25000
MH Iteration

Reference

Eddelbuettel, Dirk, Romain François, J Allaire, Kevin Ushey, Qiang Kou, N Russel, John Chambers, and D Bates. 2011. "Rcpp: Seamless R and C++ Integration." *Journal of Statistical Software* 40 (8): 1–18.

RCpp. 2018. "Rcpp for Seamless R and C++ Integration." http://www.rcpp.org/.

Stable Markets. 2018. "Speeding up Metropolis-Hastings with Rcpp." https://stablemarkets.wordpress.com/2018/03/16/speeding-up-metropolis-hastings-with-rcpp/.

Wikipedia contributors. 2018a. "C++ — Wikipedia, the Free Encyclopedia." https://en.wikipedia.org/wiki/C%2B%2B.

———. 2018b. "MCMC — Wikipedia, the Free Encyclopedia." https://en.wikipedia.org/wiki/Markov_chain_Monte_Carlo.