Critical efficiency improvements of mcmcse

by Kushagra Gupta

PROJECT INFO

Project title: Critical efficiency improvements of mcmcse
Project short title: mcmcse
url of project idea page: https://github.com/rstats-gsoc/gsoc2021/wiki/Critical-efficiency-improvements-of-mcmcse

BIO OF STUDENT

I am a final year undergraduate in the Department of Mathematics and Statistics, Indian Institute of Technology Kanpur (IIT Kanpur), with a keen interest in coding and open source development. I was the coordinator of the machine learning division of Programming Club IIT Kanpur and the mathematics and statistics society of IIT Kanpur. I have also submitted two research papers currently under review with JMLR (Gupta and Vats (2020)) and Bayesian Analysis (Gupta et al. (2021)), and have performed extensive computation and testing for the same. Working on these projects have improved my proficiency in aspects such as writing algorithms; testing, debugging, and optimizing code; porting algorithms between software languages such as R and C++; as well as documenting and presenting code. I have been at the top of my class in graduate and doctoral statistics courses at IIT Kanpur, gaining insight into relevant areas for this project through courses in Markov chain Monte Carlo (MCMC), Bayesian Analysis, statistical simulations, time series, stochastic processes, numerical linear algebra, scientific computing and data structure and algorithms. I am particularly interested in Monte Carlo algorithms, and have extensively worked in the area as part of my research work. In particular, my experience in output analysis of MCMC through my research work coupled with strong coding skills garnered through internships and projects make me a good candidate for this project.

CONTACT INFORMATION

Student name : Kushagra Gupta
Student postal address : 6/212 Vidyadhar Nagar, Jaipur, Rajasthan, India, 302023
Telephone : +917062347318, +918400198437
Webpage : https://kushagragpt99.github.io/
Emails : kushagra.gpt99@gmail.com, kushgpt@iitk.ac.in
Github handle https://github.com/kushagragpt99/
Skype ID : Kushagra Gupta

STUDENT AFFILIATION

Institution : Indian Institute of Technology Kanpur
Program : Bachelor of Science, Mathematics and Statistics
Stage of completion : Currently in my 4th and final year.
Contact to verify : Prof. G Neelakantan (gn@iitk.ac.in) / Dean, Students Affairs, IIT Kanpur (dosa@iitk.ac.in)

SCHEDULE CONFLICTS

I have been admitted into the MS Statistics program in Stanford University, and I will be pursuing courses worth half the credits of a normal semester during the summers.

MENTORS

Evaluating mentor : Prof. Dootika Vats (dootika@iitk.ac.in)
Co-mentor : Prof. James Flegal (jflegal@ucr.edu)

Last year I did my undergraduate research projects with Prof. Vats which resulted in two paper. In 2021, I am not involved with any research project/coursework with Prof. Vats. I have been in contact with the mentors and have prepared this proposal with their input.

PROJECT DESCRIPTION AND BACKGROUND

Markov chain Monte Carlo (MCMC) is a sampling-based method for estimating features of probability distributions. By constructing a Markov chain that has the target as its equilibrium distribution, one can obtain a sample of the desired distribution by recording states from the chain. MCMC methods produce a serially correlated, yet representative, sample from the desired distribution. MCMC is commonly used in Bayesian settings, but it is also useful in other situations (see eg Caffo et al. (2005), Geyer (2011)). Popular packages like `rstan`, `rjags`, `R2WinBUGS`, `mcmc` and `MCMCpack` provide efficient implementation of MCMC algorithms. The analysis of the Monte Carlo estimates from the MCMC output requires the study of Monte Carlo standard error with the following central limit theorem (CLT)

$$\sqrt{n}\left(\hat{\mu}_n - \mu\right) \xrightarrow{d} N_p\left(0, \Sigma\right),$$

where $\hat{\mu}_n$ is the ergodic average of the estimates of the parameter of interest μ. We require the estimation of Σ for output analysis through the CLT. Since the samples obtained are correlated, these quantities require more sophisticated tools than usual sample estimators.

The R package `mcmcse` (Flegal et al., 2015) provides estimates of Monte Carlo standard errors for MCMC when estimating means or quantiles of functions of the MCMC output, time series and other correlated processes.

> **Q:** *When to stop the MCMC simulation?*

The package plays an important role in answering this question by quantifying the quality of estimates after MCMC output has been obtained by the user. The package also provides univariate and multivariate estimates of effective sample size (using the `multiESS` function) and tools to determine whether enough Monte Carlo samples have been obtained (using the `minESS` function) for a specified tolerance.

The package has been dowloaded over 48,000 times and has 106 citations on Google Scholar. Currently, there are no other package that calculate multivariate effective sample size of the Markov chain. There are a few other packages in R that do univariate effective sample size calculations, the most popular of which is `coda` (Plummer et al. (2008)) . `coda` uses inconsistent parametric estimators in comparison to theoretically consistent estimators in `mcmcse`. Theoretical guarantees in `mcmcse` give it an advantage for highly correlated Markov chains.

The package `sandwich` (Zeileis et al. (2020)) provides functions for model-robust covariance matrix estimators, however the package specifically caters to heteroscedasticity- and autocorrelation-consistent (HAC) covariance matrix estimators for time series data and not for MCMC. In addition, the implementation of the spectral variance estimator `weightsAndrews(X, bw, kernel = "Bartlett", approx = "AR(1)")` in `sandwich` is orders of magnitude slower than the counterpart `mSVEfft(X, b, method = "bartlett")` in `mcmcse`. Functions in `mcmcse` package are often called on massive matrices with rows in the order of millions and columns in the order of thousands, requiring efficient and accurate implementation of the package.

In addition, the following packages on CRAN are dependent on `mcmcse` :

1. `dirmcmc`
2. `postpack`
3. `sklarsomega`
4. `eiCompare`
5. `stableGR`
6. `qbld`
7. `copCAR`
8. `nse`
9. `bpgmm`
10. `jointAI`
11. `bayes4psy`

The popular classes of estimators of the asymptotic covariance matrix Σ are:

1. Spectral variance with Bartlett window (`mcse.multi(X, method = "bartlett")`)
2. Spectral variance with Tukey window (`mcse.multi(X, method = "tukey")`)
3. (Non-overlapping) batch means (BM) (`mcse.multi(X, method = "bm")`)
4. Overlapping batch means (OBM) (`mcse.multi(X, method = "obm")`)
5. Initial sequence estimator (`mcse.initseq(X)`)

All classes of estimators account for serial correlation in the Markov chain up to a certain lag (denoted as b), called the bandwidth and batch size in spectral variance and (O)BM estimators, respectively. The choice of b is crucial to finite sample performance as a large batch size yields high variability in the estimator and a small batch size can lead to significant underestimation of Σ (see Schmeiser (1982)). Liu et al. (2018) propose a mean square error (MSE) optimal batch size, which is implemented in the package (`batchsize(X, method)`). The most common estimators of Σ in MCMC are batch means (BM) estimators, where MSE optimal batch size are proportional to $n^{1/3}$, where n is the length of the Markov chain. The technique for estimation of batch size fits a stationary autoregressive process of order m (`ar(x, aic = TRUE)`) to approximate the marginals of the Markov chain $\{Y_t\}$, which yields a closed form expression for the unknown proportionality constant. The order of the AR process is determined by Akaike information criterion (AIC).
For $R(k) = E_F[(Y_t - \theta)(Y_{t+k} - \theta)^T]$, where $\theta = E_F[Y_t]$, define

$$\Gamma^{(q)} = -\sum_{k=1}^{\infty} k^q \left[R(k) + R(k)^T\right].$$

mcmcse follows Liu et al. (2018) to estimate the MSE optimal batch size by

$$b_{opt} \propto \left(\frac{\sum_{i=1}^{p} (\Gamma_{i,i}^{(1)})^2}{\sum_{i=1}^{p} \Sigma_{i,i}^2} \right)^{1/3} n^{1/3}, \tag{1}$$

where p is the dimension of the Markov chain and the proportionality constant is known and depends on the choice of variance estimator. This approach is similar to Andrews (1991), which has the batch size calculation that we need to implement as part of this project. The method used in the package for batch size estimation is given by the following algorithm 1-

Algorithm 1: Batch size estimation in mcmcse

Data: A Harris F-ergodic Markov chain $\{X_t\}$ and $Y_t = g(X_t)$.
Result: MSE optimal batch size

1 Fit an AR(m) model for each marginal of the Markov chain, where m is determined by Akaike information criterion : `ar.fit = = ar(x, aic = TRUE)`.
2 Let $\gamma(k)$ be the lag k autocovariance function and $\phi_1, ..., \phi_m$ be the autoregressive coefficients. The autocovariances $\gamma(k)$ are estimated by the sample lag autocovariances, $\hat{\gamma}(k)$: `gammas <- as.numeric(acf(x, type = "covariance", lag.max = ar.fit$order, plot = FALSE)$acf)`.
3 Estimate ϕ_i and the variance of the stochastic component of the process (σ_e^2) by $\hat{\phi}_i$ and $\hat{\sigma}_e^2$, by solving the Yule-Walker equations : `phi <- ar.fit$ar`, `sigma2 <- ar.fit$var.pred`
4 For the *ith* component of the Markov chain, the resulting AR(m)-fit estimators are given by equation 2 and 3.
5 Substitute the expressions in 1 with the appropriate proportionality constants (refer Liu et al. (2018)).

Define the estimators of Σ and Γ of an AR(m) process by

$$\Sigma_{p,i} = \frac{\hat{\sigma}_e^2}{(1 - \sum_{i=1}^{m} \hat{\phi}_i)^2} \tag{2}$$

$$\Gamma_{p,i} = -2 \left[\left(\sum_{i=1}^{m} \hat{\phi}_i \sum_{k=1}^{i} k\hat{\gamma}(k-i) \right) + \frac{\hat{\sigma}_e^2 - \hat{\gamma}(0)}{2} \left(\sum_{i=1}^{m} i\hat{\phi}_i \right) \left(\frac{1}{1 - \sum_{i=1}^{m} \hat{\phi}_i} \right) \right] \tag{3}$$

The function `ar(X, method)` provides additional methods to fit the AR model (like OLS and MLE) and estimate autoregression coefficients for the fitted model.

Reporting $p \times p$ covariance matrix estimates is impractical and uninterpretable, especially for high dimensional Markov chains. The motivation of estimating Monte Carlo standard error is to ensure that said error is small. This is essentially the idea behind estimating effective sample size and ensuring that the estimated effective sample size is larger than a pre-specified lower bound. The package mcmcse is the only package that performs multivariate effective sample size calculations. The function `minESS(p, alpha, eps)` calculates the minimum effective sample size required for a specified relative tolerance level (ϵ). This is extremely useful as it provides users with an indication of the length of the Markov chain that is required for a particular tolerance , thus providing for accurate

assessment of required computation power and warning against premature termination. The minimum effective samples required when estimating a vector of length p, with $100(1-\alpha)\%$ confidence and tolerance of ϵ is

$$\texttt{minESS} \geq \frac{2^{2/p}\pi}{(p\Gamma(p/2))^{2/p}} \frac{\chi^2_{1-\alpha,p}}{\epsilon^2}.$$

In addition, the function `multiESS(X)` calculates the effective sample size of the Markov chain, using the multivariate dependence structure of the process.

$$\texttt{multiESS} = n\frac{|\Lambda|^{1/p}}{|\Sigma|^{1/p}},$$

where Λ is the sample covariance matrix and Σ is an estimate of the Monte Carlo standard error.

CODING PLAN AND METHOD

Improve `batchsize()`

The function `batchsize` calculates the bandwidth or the batch size (denoted by b) in spectral variance and (O)BM estimators. The importance of b to the quality of estimation of the covariance matrix is described in the previous section, and is an important subroutine for many functions of the package. Figure 1 shows the profile for estimating Σ for a 100 dimensional Markov chain (henceforth denoted as $\{X_t\}$) of length 10^5. Calculation of the output of `batchsize` (henceforth denoted by b_{opt}) takes 2580 milliseconds, out of the total 2880 milliseconds for running `mcse.multi`. This shows how `batchsize` is a prime bottleneck in the estimation of Σ, an effect compounded for large matrices.

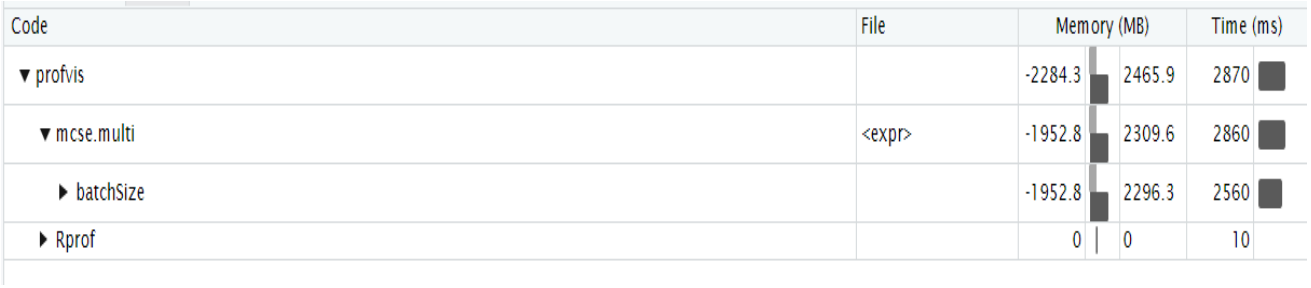
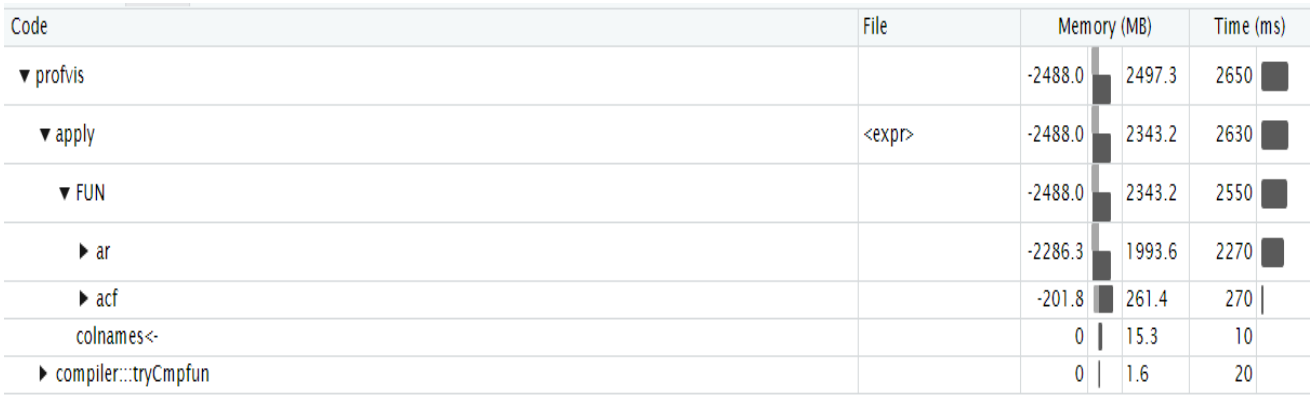


Figure 1: Profile for `mcse.multi`

As explained in the previous section, calculating b_{opt} requires fitting a stationary autoregressive process to approximate the marginals of the Markov chain. The package already utilizes the fast `ar` implementation for the task. In the existing implementation, `batchsize` calls `apply` on each component of the Markov chain for AR approximation. Profile of the code for `batchsize` on $\{X_t\}$ in figure 2 shows that this is the computationally heaviest component of the code, as the `arp_approx` function is being called on every component of the chain. As part of this task, I propose to try two approaches to speed up `batchsize`

Approach 1

I will implement the `batchsize` function in `Rcpp` to speed up the loop over the components of the Markov chain. I will still use the `ar` implementation within `Rcpp` because of its



Code	File	Memory (MB)		Time (ms)
▼ profvis		-2488.0	2497.3	2650
▼ apply	<expr>	-2488.0	2343.2	2630
▼ FUN		-2488.0	2343.2	2550
▶ ar		-2286.3	1993.6	2270
▶ acf		-201.8	261.4	270
colnames<-		0	15.3	10
▶ compiler:::tryCmpfun		0	1.6	20

Figure 2: Profile for `batchsize`

efficiency. The implementation of the component of the code to substitute `apply` will look similar to :

```
RObject arp_approx(x) {
  Rcpp::Environment base("package:stats");
  Rcpp::Function ar = base["ar"];
  ar.fit = ar(x, aic = TRUE)
  ...  // formulas to calculate autoregression coefficients and an
  // estimate of the portion of the variance of the time series that is
  //  not explained by the autoregressive model (equation (2) and (3)).
}

uword batchsize(x, method = "bm", g = NULL) {
  ...
  // chain denotes the n * p dimensional Markov chain
  // ar_fit is an RObject to store the output of arp_approx()
  for (uword i = 0; i < size(chain)[1]; i++)  {
      ar_fit[i,] = arp_approx(chain[,i])
  }
  ... formulas to calculate the appropriate proportionality constants
// (refer to algorithm 1)
}
```

Approach 2

I will use the `doParallel` package for parallel application of `arp_approx` across components. As fitting an AR approximation on each component is independent of the other components, we can fit the approximation parallelly. The implementation of the component of the code to substitute `apply` will look similar to :

```
library(doParallel)
no_cores <- detectCores() - 1
cl <- makeCluster(no_cores, type="FORK")
registerDoParallel(cl)
```

```
batchSize <- function(x, method = "bm", g = NULL) {
    ...
    # chain denotes the n * p dimensional Markov chain
    # arp_approx function is the same as the existing function, as it
    # is an efficient implementation in R

    ar_fit <- foreach(i=10:10000) \%dopar\% arp_approx(chain[,i])^2
    ... remaining implementation is the same as the existing one.
  }
```

I will write tests to compare the performance of the two approaches and the final implementation will include the most efficient approach for the task. The tasks will focus on comparing performance across parameters like :

1. Running time of the approach.
2. Memory consumption of the approach.
3. Easy integration into the existing code structure.

As the marginals of the Markov chain themselves may not follow a Markovian structure, the order of the AR process is not known. Therefore, we use the Akaike Information Criterion (AIC) to choose the order of the autoregressive model. This step is expensive. Figure 3 and 4 compare the profiles when `aic` is set as TRUE or FALSE. The subroutine takes double the time when it needs to choose the order of the autoregressive model, and this latency stacks up in high dimensional Markov chains because the order needs to be estimated for each component.

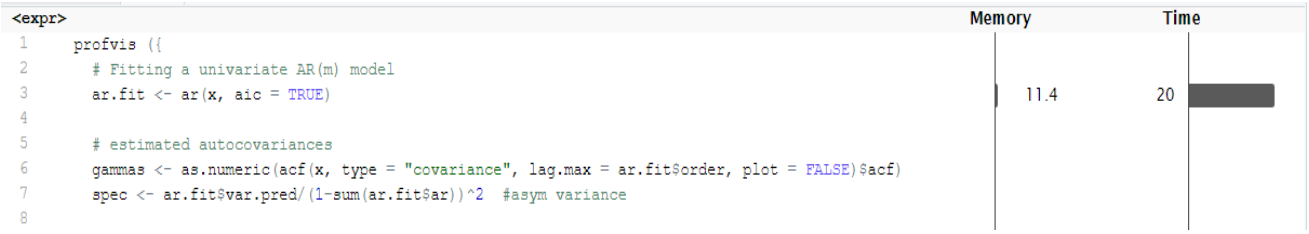


Figure 3: Profile for `arp_approx` with `aic = TRUE`

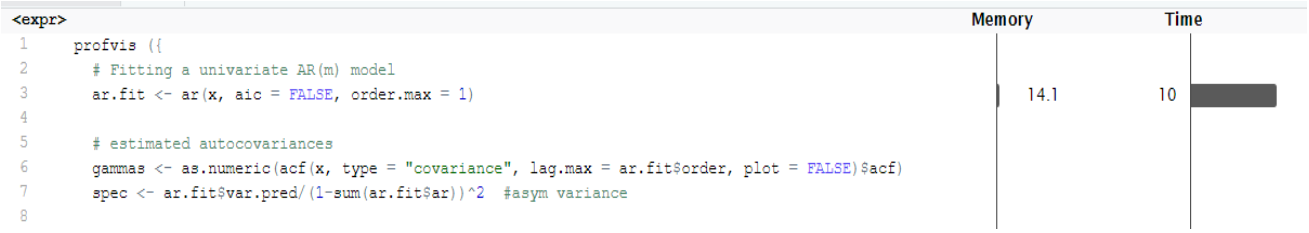


Figure 4: Profile for `arp_approx` with `aic = FALSE, order.max = 1`

One possible workaround to this problem is to fix the order to a prespecified value. The package `sandwich` in its implementation of `weightsAndrews` also estimates the optimal batch size by fitting autoregressive models to the marginals. However, it offers just two options : AR(1) and ARMA(1,1) as choices. I plan to expand upon this implementation by adding an option to fix the order of the autoregressive model to any pre-specified value. The existing implementation in `mcmcse` (which includes estimating the order)

is already significantly faster than sandwich. This addition will allow the package to comprehensively outperform sandwich while improving the performance of the package. The implementation of the component of the code will look similar to :

```
arp_approx <- function(x, choose_order = TRUE, order = NULL)  {
  ar.fit <- ar(x, aic = choose_order, order.max = order)
  ... remaining implementation remains the same.
}
```

I also propose to add tests to compare the drop in the quality of estimates when fixing the order to a pre-specified value over estimating the order at every turn.

The ar function provides multiple methods to fit autoregressive models, including fitting the Yule-Walker equations, MLE and OLS. Currently, the package defaults to using the Yule-Walker equations to estimate the parameters. I will compare the performance of other methods in ar to choose the most optimal (in terms of speed of computation and estimate quality) method for estimating parameters.

Develop new `batchsize()`

Andrews (1991) defines a more general algorithm for batch size calculations. A brief overview of the steps are described in the following algorithm :

Algorithm 2: Batch size estimation in Andrews (1991)

Data: A Harris F-ergodic Markov chain $\{X_t\}$ and $Y_t = g(X_t)$.
Result: MSE optimal batch size

1. Specify p univariate approximating parametric models or a single multivariate approximating parametric model for $\{Y_t\}$.
2. Estimate the parameters of the approximating parametric model(s) by standard methods.
3. Substitute these estimates into a formula (equation 4) that expresses $\alpha(q)$ as a function of the parameters of the parametric model(s).
4. The value of q is decided by the asymptotic covariance matrix estimator. The batch size is given by $\alpha(q)n^{1/(2q+1)}$.

$$f^{(q)} = -\sum_{k=1}^{\infty} k^q \left[R(k) + R(k)^T\right].$$

$$\alpha(q) = \frac{\sum_{i=1}^{p} w_i (f_{ii}^{(q)})^2}{\sum_{i=1}^{p} w_i f_{ii}^2}, \tag{4}$$

where w_i are weights, and $f_{ii}^{(q)}$ and f_{ii} denote the ith diagonal element of $f^{(q)}$ and Σ respectively.

The first two steps of algorithm 2 are similar to the existing implementation in batchsize, and can be performed as :

```
ar.fit <- ar(x, aic = TRUE)
gammas <- as.numeric(acf(x, type = "covariance", lag.max = ar.fit$order,
                         plot = FALSE)$acf)
Sigma <- ar.fit$var.pred/(1-sum(ar.fit$ar))^2
```

The formula for $f^{(q)}$ will need to be derived, following the technique from section 3.1 of Liu et al. (2018). Once estimated, these values can be substituted into equation 4. The value of q will be decided based on the estimator of Σ with the information in Liu et al. (2018) and Andrews (1991). The existing implementation of `batchsize` assumes $q = 1$ for all lag windows, which is not the case for 'Tukey-Hanning' lag windows. Therefore, this implementation will lead to theoretically consistent MSE optimal batch sizes. A competing package `coda` for calculating the effective sample size (ess) follows an approximate approach. The implementation in `mcmcse` is theoretically backed, therefore a better implementation of `batchsize` will be useful for the package.

This implementation will face the same bottlenecks as the existing implementation of `batchsize`, therefore code development for the previous task will naturall extend to this task. In addition, I propose to keep the existing implementation as an option for users and compare the performance of the two functions. The performance will be compared for :

1. Extremely high dimensional Markov chains ($p >> n$).
2. Markov chains exhibiting high correlation.
3. Markov chains with large length ($n >> p$).
4. Markov chains with known Σ.

In addition, I will compare the running time and memory consumption of the two functions.

Create `mcse` object class

The output of `mcse.multi` is a list containing information about the Monte Carlo standard errors in the Markov chain. This list can be converted into a `mcmcse` class by specifying the `class` attribute. If `blather = TRUE` in `mcse.multi`, the list has the asymptotic covariance matrix (`sig.mat`), mean estimate (`mu.hat`), length of Markov chain (n), method used for estimating Σ (`method.used`), batch size (b), Boolean variable for if the matrix is adjusted (`Adjustment-used`), and some additional messages (`messages`) as its components.

After converting the `mcse.multi` output to a class, I propose to convert the existing functions to methods for the class. Currently, the following functions can be directly converted to methods for the class `mcmcse` :

1. `confRegion(mcse.obj, which = c(1,2), level = .95)`
2. `adjust_matrix(mat, N, epsilon = sqrt(log(N)/dim(mat)[2]), b = 9/10)`
3. `qqTest(mcse.obj)`

In addition, the output of `mcse.initseq` and `mcse.q` needs to be modified to be consistent with `mcse.multi`, after which their output can be made identifiable with the `mcmcse` class.

Making a class for the output from the package would make it easier to be used by external packages, potentially benefiting the reach and utility of `mcmcse`. Additionally, as this package will serve as the foundation for a user-oriented package for Simulation Output Analysis, making the output identifiable with a class would make the integration of the package easier.

Rcpp sugar

`profvis` doesn't support profling of individual functions in `Rcpp`. I propose to use the profiling tools in visual studio for `cpp` functions to identify computationally heavy parts and convert them into sugar functions. I will specifically focus on implementations in

`mcse.multi` as they are the most commonly called functions of the package. Efficiency tests aimed at the performance for high dimensional matrices can be run in `R`. Further tests will be decided in consultation with the mentors.

Numerical Instabilities

The methods used for estimating the standard error are consistent and eventually converge to the correct value. But since we are performing a finite approximation of the estimates, the expressions for the estimators might give mathematically inconsistent results. A classic example of this is some eigenvalues of the variance estimate being negative to give non positive definite estimates and possibly a negative determinant. The user needs to be alerted in such a case to provide more input so that consistency eventually kicks in and we get mathematically consistent results.

In cases where the sample size is low or the Markov chain exhibits high correlation, finite-sample estimate of Σ may not be positive definite. Additionally, the package returns the lugsail estimates (see Vats and Flegal (2018)) to counter the significant negative bias in the estimate. The lugsail estimates involve subtracting estimates of Σ for different values of b (see equation 5). Although the lugsail estimates eventually converge to the truth, for small sample sizes the difference of the two estimates of Σ may lead to covariance matrices that are not positive definite. Cases like these need to be dealt with. Currently the package uses the `adjust_matrix` function to increase the eigenvalues if the estimated Σ is not positive definite. I propose to use the `nearPD` function from the `Matrix` package instead of `adjust_matrix`. `nearPD` implements the algorithm of Higham (2002) to compute the nearest positive definite matrix, and then forces positive definiteness using code from `posdefify`.

$$\widehat{\Sigma}_{lugsail} = \frac{1}{1-c}\widehat{\Sigma}_b - \frac{c}{1-c}\widehat{\Sigma}_{b/r}\,, \tag{5}$$

(where $\widehat{\Sigma}_{lugsail}$ denotes the lugsail batch means estimates, and $\widehat{\Sigma}_k$ denotes the batch means estimate with $b = k$).

In addition, determinants in the package are calculated as the product of eigen values. This might be unstable if some of the eigen values are extremely small or large. This can be corrected by modifying the function into exponential of sum of log of eigen values. The code for this would look similar to :

```
  det = exp(sum(log(eigen(mat, only.values = TRUE)\$values)))
```

Benchmarking and Profiling

I propose to use the package `profvis` for visualizing code profiling data from R. To run code with profiling, wrap the expression in `profvis()`. By default, this will result in the interactive profile visualizer opening in a web browser. Example implementation :

```
  profvis({
  g <- ggplot(diamonds, aes(carat, price)) + geom_point(size = 1,
              alpha = 0.2)
  print(g)
})
```

Profiling for Rcpp

I will use the profiling tools in visual studio for `cpp` functions as `profvis` doesn't support profiling of individual functions in `Rcpp`.

Profiling memory

I plan to create a *.Rmd* file while optimizing code to store all the approaches for performance optimization, which can be used as a reference for similar problems in the future. I propose to use the `microbenchmark` package (which provides an infrastructure to measure and compare the execution time of R expressions accurately for small and specific chunks of code) to compare the performance of different approaches.

Possible approaches to improving the performance of bottlenecks

- If the bottleneck is a function in a package, it's worth looking at other packages that do the same thing. Two good places to start are:
  1. CRAN task views: If there's a CRAN task view related to the problem domain, it's worth looking at the packages listed there.
  2. Reverse dependencies of `Rcpp`, as listed on its CRAN page. Since these packages use `cpp`, it's possible to find a solution to the bottleneck written in a higher performance language.
  3. For Google, trying `rseek` and for StackOverflow, restricting the search by including the R tag, [R], in the search.
- Knowing the exact format of the input so that appropriate, more efficient functions can be used instead of general functions. For example, `as.data.frame()` is quite slow because it coerces each element into a data frame and then `rbind()`s them together. If you have a named list with vectors of equal length, you can directly transform it into a data frame. In this case, if you're able to make strong assumptions about your input, you can write a method that's about 20x faster than the default.

I plan to perform benchmarking using the package `rbenchmark`. Given a specification of the benchmarking process (counts of replications, evaluation environment) and an arbitrary number of expressions, `benchmark` function in `rbenchmark` evaluates each of the expressions in the specified environment, replicating the evaluation as many times as specified.

Correcting error or warning messages

The current implementation of the package throws error or warning messages at corner cases. I plan to identify possible modifications that correct for these case without breaking the implementation. I have listed possible modifications for some of the messages, which will be implemented in consultation with the mentors :

1. Line 339 in `mcse.q`

```
      if (! is.numeric(size) || size < 1 || size == Inf)
            stop("'size' must be a finite numeric quantity larger than 1.")
```

   Solution : Convert error to a warning and set $b = \sqrt{n}$ instead of the variable size.

2. Line 348 in mcse.q

```
if (! is.numeric(q) || q <= 0 || q >= 1)
  stop("'q' must be from (0, 1).")
```

Solution : Convert error to a warning message and set $q = 0.5$.

3. Line 153 in mcse.multi

```
if(method != "bm" &&  method != "obm" && method != "bartlett" &&
   method != "tukey")
{
  stop("No such method available")
}
```

Solution : Convert error to a warning message and set method to a default value (like "bm").

4. Line 159 in mcse.multi

```
if(r < 1) stop("r cannot be less than 1")
```

Solution : Convert error to a warning message and set $r = 3$.

5. Line 196 in mcse.multi

```
if (!is.numeric(size) || size < 1 || size >= n || floor(n/size) <=1)
  stop("size is either too large, too small, or not a number")
```

Solution : Convert error to a warning and set $b = \sqrt{n}$ instead of the variable size.

6. Line 216 in mcse.multi

```
if(floor(b/r) < 1) stop("Either decrease r or increase n")
```

Solution : Convert error to a warning message and set $b = r$.

Solution to point 1 and 5 is chosen to give a balance between batch size and number of batches. In the solution to point 2, we default to the median. In point 3, we default to the computationally easiest method which is batch means ("bm"). Solution to point 5 chooses the most commonly used value of r in lugsail which is 3. Problem in point 6 arises when the input Markov chain is fast mixing. In such a scenario, the estimator requires a small lag to capture the correlation in the Markov chain, resulting in a small value of b. For example, the following code gives the optimal batch size as 1 for iid normal data.

```
set.seed(10)
x = rnorm(1e5)
batchsize(x)
## 1
```

In such scenarios, small values of b are adequate for the estimator. Therefore we set b to its minimum possible value, i.e r, which allows for the estimation of lugsail version of the estimator (lugsail estimator estimates Σ with batch size b and b/r).

Computationally heavy coding with Rcpp

The function `mcmse.q` in `mcmcse.R` performs all computation for different methods in `R`. Refactoring these functions to perform the computation in `cpp`, following the code structure similar to `mcse.multi`, will increase the efficiency of `mcmse.q`. This change will involve writing `cpp` functions for batch means (`mcse.q(method = "bm")`), overlapping batch means (`mcse.q(method = "obm")`), and subsampling bootstrap (`mcse.q(method = "bm")`). Functions `minESS` and `multiESS` can also be implemented in `Rcpp` if deemed necessary.

Documentation

I plan to use the package `roxygen2` to maintain and update the documentation. `roxygen2` generates *.md* files and creates `NAMESPACE` with `R` and `Rcpp`. The package requires adding roxygen comments to the source code (similar to the functions in `mcmcse.R`). I plan to add the necessary roxygen comments to all the remaining and the new functions to bring uniformity in all documentation. I will add the comments while working on the function to maintain documentation while I am working on the function. Currently, the package requires roxygen comments for the following files :

- `confRegion.R`
- `mcse.R`
- `mcse_initseq.R`
- `mcse_multi.R`
- `minESS.R`
- `multiESS.R`
- `qq_test.R`
- `inseq.cpp`
- `lag.cpp`
- `mbmc.cpp`
- `mobmc.cpp`
- `msvec.cpp`

Testing

Previous versions (version 1.3) of the package used `testthat` for user testing and code testing. I propose to expand `test_validity.R` from version 1.3 of the package for the remaining methods like obm and add new tests for new functions implementated during the coding period. I will also add tests for `confRegion` in `confRegion.R`, `mcse.q` in `mcmcse.R`, `mcse.initseq` in `mcse_initseq.R`, `batchsize` in `mcse_multi.R`, `minESS` in `minESS.R` and `multiESS` in `multiESS.R`.

Demo for the package

A demo for the package will serve to show users how to perform output analysis for MCMC by weaving together the functions of the package. I propose to create a demo that will help the existing vignette of the package and assist the users uninitiated with the theory and common practices in output analysis for MCMC. I plan to setup the code to be able to identify the demo created for the package and use the `demo` function in the `utils` package to call the demonstration R scripts.

Minor tasks

I have identified some minor tasks along with those listed on the GSoC project page which will improve the package. The tasks are listed below :

1. At present, the package identifies non positive definite matrices by checking if the eigen values are positive. The current implementation calculates all the eigen values for this task using the `eigen` function. A more efficient solution would be to calculate just the **minimum eigen value** using `eigs` function from the `RSpectra` package.

2. During profiling, if the **matrix calculation** functions are found to be computationally burdensome then information from Tsagris and Papadakis (2018) and Tsagris et al. (2016) can be used to speed up the calculations.

3. The current implementation utilizes a lot of `if`s to identify individual cases. This makes the code unclean and patchy. I plan to **streamline the code** so that the various options in the functions can be used without individual sub-parts for each option, to make the code clean and consistent.

4. **Lugsail** version of the estimators of Σ is the default option in the current implementation, but this is not reflected in the documentation. I propose to make necessary changes to the documentation and add a new argument `lug_params = c(r,c)` to the functions which estimate Σ.

5. The file `mcse.R` is redundant as it calculates the multivariate BM and MSV estimator for the covariance matrix in `R`, while the `Rcpp` implementation already exists in `mcse.multi()`. Therefore, this file can be removed.

6. The current implementation of `mcse.q` does not provide the option for calculating the MSE **optimal batch size**. This option can be added easily from `mcse_multi.R`.

7. Output of `mcse.initseq` needs to be made consistent with `mcse.multi`. This would allow the output to be identifiable with `mcmcse` class.

8. As the package deals with MCMC, a **Bayesian example** would sever better in the documentation. I propose to modify the examples in the documentation from MAR to a Bayesian example.

MANAGEMENT OF CODING PROJECT

The package will be version controlled with git. As proposed in my coding plan, I will use `testthat` for user and code testing. Using `testthat` will also ensure code submission. The coding and documentation style will follow the Tidyverse style guide. I will make multiple minor commits adhering to each new functionality, followed by a major commit every week to conclude the successful completion of the task allocated to that week. The minor commits will be helpful for the mentors to track my progress over a week and will dictate the schedule of meetings. My major commits will track my workflow over the entire GSoC period. Missing out on major commits for two continuous week would indicate a problem. I plan to have weekly (or biweekly upon the discretion of the mentors) zoom meetings with my mentors to communicate my progress and discuss possible issues.

TIMELINE

PERIOD	TASK
Community bonding period	Research scientific literature for optimal batch size and study efficient implementation to fit AR models. Get familiar with the tools mentioned in the proposal. Discuss some of the roadblocks and plans mentioned in the proposal with mentors.
June 7 - June 14	Implement Approach 1 to improve batchsize(). Write testthat, check for numerical instability and profiling, and write roxygen comments.
June 14 - June 21	Implement Approach 2 to improve batchsize() and write comparison tests for the two approaches.Write testthat, check for numerical instability and profiling, and write roxygen comments.
June 21 - June 28	Implement the pre-specified fixed order AR fitting approach and compare performance of methods other than Yule Walker equations for AR parameter fitting. Write testthat, check for numerical instability and profiling, and write roxygen comments.
June 28 - July 5	Implement the new batchsize() function from Andrews. Write testthat, check for numerical instability and profiling, and write roxygen comments.
July 5 - July 12	Implement mcse.q and ESS functions in Rcpp. Write testthat, check for numerical instability and profiling, and write roxygen comments.
July 12 - July 19	Clean-up for phase 1 evaluation. Create mcse object class.
July 19 - July 26	Profiling of Rcpp code and shifting computationally heavy parts to rcpp sugar. Write testthat, check for numerical instability and profiling, and write roxygen comments.
July 26 - Aug 2	Complete the Minor Tasks listed in CODING PLANS. Write testthat, check for numerical instability and profiling, and write roxygen comments.
Aug 2 - Aug 9	Implement corrections for error or warning messages identified in CODING PLANS. Write testthat, check for numerical instability and profiling, and write roxygen comments.
Aug 9 - Aug 16	Implement the demo scripts and a Bayesian running example for the documentation. Write roxygen comments for the untouched functions listed in documentation section of CODING PLANS.
Aug 16 - Aug 23	Final eval prep and cleanup - tying any loose ends up to this point (bug fixes, documentation, incomplete implementation, remaining numerical instability) to submit work for evaluation.

TESTS

Code for solution to the tests.
Code link to similar problems on MCMC.

Test 1 (easy)

1. Download the mcmcse package from CRAN and use the function ess on a vector foo of length 1e4 randomly drawn from a standard normal distribution.
2. Make a random matrix of size 10 x 10 and produce only the eigenvalues of the matrix.

Solution : ESS is the size of an iid sample with the same variance as the current sample. ESS is given by

$$ESS = n\left(\frac{|\Lambda|}{|\Sigma|}\right)^{1/p},$$

where Λ is the sample variance, and Σ is an estimate of the variance in the Markov chain central limit theorem. Code chunks relevant for this task are :

```
#Sampling 1e4 samples from N(0,1) & using ess function
foo <- rnorm(1e4)
ess(foo)

#Sampling a random matrix and calculating eigen values
rmatrix <- matrix(rnorm(100), ncol = 10)
eigen(rmatrix, only.values = TRUE)\$values
```

Test 2 (medium)

Implement an efficiency profile of the `batchSize` function using `profvis`. Do this for varying sizes of input matrices.

Solution : This task involved constructing various Markov chains and calling `batchSize` on them. In my solution, I used the following chains -

1. AR(1) process of length 10^5.
2. Markov chain constructed for the hard task (i.e. MCMC output targeting a 100 dimensional standard normal Gaussian distribution) of length 10^5.
3. VAR(1) of dimension 2 and length 10^5.
4. VAR(1) of dimension 2 and length 10^6.
5. VAR(1) of dimension 2 and length 10^7.
6. VAR(1) of dimension 500 and length 10^5.

The results for this task shows how `batchSize` acts as a bottleneck in `mcse.multi` while being an integral subroutine for many other functions of the package. The execution time and memory consumed increase by factors of 10 if the length of the Markov chain is increased by the same factor. Additionally, both the execution time and memory blow up when the dimensions of the Markov chain are increased. This clearly shows a need of improvement in the implementation of `batchSize`.

The results of `profvis` are summarized in the following table :

Markov chain	Memory (MB)	Time (ms)
AR(1) $n = 10^5$	19.5	30
VAR(1) $p = 2, n = 10^5$	41.7	50
VAR(1) $p = 2, n = 10^6$	480.9	490
Hard task MCMC $n = 10^5$	2646.2	2740
VAR(1) $p = 2, n = 10^7$	4272.7	5890
VAR(1) $p = 500, n = 10^5$	11865.4	11990

Test 3 (hard)

1. Write a code for a random walk Metropolis-Hastings algorithm to sample from a 100 dimensional standard normal Gaussian distribution. Focus on efficient implementation of this code.
2. Calculate the effective sample size as described in this paper in a way that is numerically stable, and does not utilize any inbuilt functions. Make sure you write your own function for this.

Solution : Random walk Metropolis-Hastings (RWMH) is an MCMC algorithm commonly used for high dimensional target distributions. The algorithm involves sampling from a proposal distribution and accepting it with acceptance probability (α). If the proposed value is rejected, then the previous value of the Markov chain is accepted as the most recent update. The code for RWMH function for the task is :

```
RWMH <- function(n, init, h)  {
   p = length(init)
   accept = 0
   output = matrix(, nrow = n, ncol = p)
   output[1,] = init
   h_root = sqrt(h)
   for(t in 2:n) {
     prop =  output[t-1,] + rnorm(p,0,h_root)
     log_ratio = log_unnormalised_posterior(prop) -
                 log_unnormalised_posterior(output[t-1,])
                 # work with log for numerical stability
     if(log(runif(1)) < log_ratio) {
       output[t,] = prop
       accept = accept + 1
     }
     else  {
       output[t,] = output[t-1,]
     }
   }
   print(accept/n) # acceptance probability
   output
}
```

We work with log of the posterior probability for numerical stability as extremely small values may not be captured in R. Additionally, we require small values of the step size (h)

in the proposal to get the sampler to explore. The ratio of probabilities ($f(y)/f(x)$) tends to 0 for proposals with a large h (for eg probability is 10^{-40} at mean and 10^{-62} after one step with h = 1). This leads to $\alpha(x, y) \sim 0$. Intuitively, the proposal at every step needs to agree on all 100 dimensions of the target, i.e. the proposal will be rejected even if it moves to an area of low probability for just 1 component of the proposal. Having a large value of h increases the chances of the Markov chain moving to unfavourable areas for any dimension. Choosing small values of h (0.1 to 0.5) increases the acceptance probability as values closer to the existing state (which was accepted and therefore doesn't have a very low probability value) will now be proposed and therefore, have a probability value that is not significantly different from the value at the previous state. Sampling from a high dimensional distribution requires proposals to agree on all dimensions of the target, which can be ensured with proposals being close to previously accepted, and therefore verified values.

The paper defines a multivariate effective sample size (ESS) that takes into account the inherent variability of the problem. As describes in the solution for the easy task, ESS is the Monte Carlo error relative to the generalized variance. Therefore, if the problem has high variability (and therefore large $|\Lambda|$) then it has a larger ESS than a problem with low variability with the same estimate of the Monte Carlo error (captured by $|\Sigma|$). Further, a multivariate ESS captures the cross-correlation across components of the Markov chain. Our definition of ESS requires a strongly consistent estimator of Σ, for which we implement the batch means estimator. The code chunk for BM estimator is :

```
# Estimate CLT covariance matrix
bm_estimator <- function(chain) {
  n = dim(chain)[1]
  p = dim(chain)[2]
  a = floor(sqrt(n)) # no of batches is sqroot(n)
  b = a
  X = make_batch(chain, n, a, b) # divide the chain into batches
  chain_mean = colMeans(X)
  bm = (t(X - chain_mean) %*% (X - chain_mean)) * (b/(a-1)) # formula for
       # batch means from Vats et al 2017
  bm
}
```

The function for calculating the ESS is :

```
# Estimate effective sample size
ess <- function(chain)  {
  n = dim(chain)[1]
  p = dim(chain)[2]
  sample_var = cov(chain) # Sample covariance matrix
  CLT_var = bm_estimator(chain) # CLT covariance estimate
  if (min(eigen(CLT_var, only.values = TRUE)\$values) <= 0)  {
    CLT_var = adjust_matrix(CLT_var, N=n) # Tweak the matrix if BM
                                          # estimate is not pd
  }
  sample_var_det = exp(sum(log(eigen(sample_var, only.values = TRUE)
                   \$values))/p) # Det ^ 1/p
```

```
  CLT_var_det = exp(sum(log(eigen(CLT_var, only.values = TRUE)\$values))/p)
                # Using exp(log(sum(eigen))) for numerical stability.
  ess = n*(sample_var_det/CLT_var_det) # ESS formula from Vats et al 2017
  ess
}
```

The determinant in the above function is calculated as the exponential of sum of log of eigen values for numerical stability. In addition, we need the `adjust_matrix` function to make covariance matrix estimates positive definite when they are not. The function increases the eigen values if they are below a threshold (epsilon) to ensure numerical stability. The code for the function is :

```
# Modify eigen values if they are below epsilon
adjust_matrix <- function(mat, N, epsilon=sqrt(log(N)/dim(mat)[2]), b=0.9)
{
  mat.adj <- mat
  adj <- epsilon*N^(-b)
  vars <- diag(mat)
  corr <- cov2cor(mat)
  eig <- eigen(corr)
  adj.eigs <- pmax(eig\$values, adj)
  mat.adj <- diag(vars^(1/2))%*% eig$vectors %*% diag(adj.eigs) %*%
             t(eig\$vectors) %*% diag(vars^(1/2))
  return(mat.adj)
}
```

Bibliography

D. W. Andrews. Heteroskedasticity and autocorrelation consistent covariance matrix estimation. *Econometrica*, 59:817–858, 1991. [p4, 8, 9]

B. S. Caffo, W. Jank, and G. L. Jones. Ascent-based Monte Carlo EM. *Journal of the Royal Statistical Society*, Series B, 67:235–251, 2005. [p2]

J. M. Flegal, J. Hughes, and D. Vats. *mcmcse: Monte Carlo Standard Errors for MCMC*. Riverside, CA and Minneapolis, MN, 2015. R package version 1.1-2. [p2]

C. J. Geyer. Introduction to Markov chain Monte Carlo. In S. Brooks, A. Gelman, X.-L. Meng, and G. L. Jones, editors, *Handbook of Markov Chain Monte Carlo*. Chapman & Hall, Boca Raton, 2011. [p2]

K. Gupta and D. Vats. Estimating Monte Carlo variance from multiple markov chains, 2020. [p1]

K. Gupta, D. Vats, and S. Chatterjee. Bayesian equation selection on sparse data for discovery of stochastic dynamical systems, 2021. [p1]

N. J. Higham. Computing the nearest correlation matrix—a problem from finance. *IMA Journal of Numerical Analysis*, 22(3):329–343, 2002. doi: 10.1093/imanum/22.3.329. [p10]

Y. Liu, D. Vats, and J. M. Flegal. Batch size selection for variance estimators in mcmc, 2018. [p3, 4, 9]

M. Plummer, N. Best, K. Cowles, and K. Vines. coda: Output analysis and diagnostics for MCMC. *R package version 0.13-3, URL http://CRAN. R-project. org/package= coda*, 2008. [p2]

B. W. Schmeiser. Batch size effects in the analysis of simulation output. *Operations Research*, 30:556–568, 1982. [p3]

M. Tsagris and E. Papadakis. Taking r to its limits: 70+ tips. *PeerJ Prepr.*, 6:e26605, 2018. [p14]

M. Tsagris, M. Papadakis, I. Tsamardinos, M. Fasiolo, B.-B. Giorgos, and J. Maintainer. Rfast reference manual. 08 2016. doi: 10.13140/RG.2.1.3722.0085. [p14]

D. Vats and J. M. Flegal. Lugsail lag windows and their application to MCMC. *ArXiv e-prints*, 2018. [p10]

A. Zeileis, S. Köll, and N. Graham. Various versatile variances: An object-oriented implementation of clustered covariances in R. *Journal of Statistical Software*, 95(1):1–36, 2020. doi: 10.18637/jss.v095.i01. [p3]