TUM SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Efficient Bayesian Inference of Hydrological Model Parameters: Mathematical Analysis and Implementation of Markov Chain Monte Carlo Approaches

Stefan Stöckl

TUM SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Efficient Bayesian Inference of Hydrological Model Parameters: Mathematical Analysis and Implementation of Markov Chain Monte Carlo Approaches

Effiziente Bayessche Inferenz von Hydrologischen Modell Parametern: Mathematische Analyse und Implementation von Markow-Ketten-Monte-Carlo Ansätzen

Author:	Stefan Stöckl
Supervisor:	Hans-Joachim Bungartz
Advisor:	Ivana Jovanovic Buha
Submission Date:	03.06.2025

I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Langenbach, 03.06.2025

Acknowledgments

I want to thank my advisor, Ivana Jovanovic Buha, for taking the time to answer my questions, giving me helpful advice, and introducing me to this topic. I also want to thank my family for their support and for proofreading this thesis.

Abstract

In this thesis, we estimate the posterior distributions of the HBV-SASK hydrological model's input parameters using Bayesian inference through Markov Chain Monte Carlo (MCMC) algorithms. Using the mean or mode of the resulting parameters' posterior distribution, the model will likely closely simulate the observed streamflow values. We describe three MCMC algorithms – the Metropolis-Hastings (MH) algorithm, the Delayed Rejection Adaptive Metropolis (DRAM) algorithm, and the Differential Evolution Adaptive Metropolis (DREAM) algorithm – as well as simple parallelizations for the first two algorithms. To analyze these algorithms mathematically, we explain the root mean square error and mean absolute error as error functions, the Monte Carlo Standard Error, Effective Sample Size and autocorrelation, as well as the $\hat{R}$ diagnostics for convergence analysis. We examine the three MCMC algorithms with these diagnostics combined with trace plots, histograms, and other diagrams. We first determine appropriate parameter values for each algorithm according to these diagnostics by evaluating them with the HBV-SASK hydrological model on the Banff river basin. Then, we use each MCMC algorithm with those parameters to obtain parameters for the HBV-SASK model trained on additional training datasets. Afterwards, we run the HBV-SASK model with those trained parameters on different evaluation sets of the Banff and Oldman river basins.

Overall, all algorithms perform similarly well regarding the error functions when evaluated on similar datasets to the training datasets. The parameters we obtained through DREAM perform worse than those of the other two MCMC algorithms if we use them on evaluation sets with different behaviors compared to the respective training set. However, the DREAM algorithm performs far better for the Effective Sample Size and autocorrelation, as well as the convergence diagnostic than the other two algorithms. The DRAM algorithm exhibits better and more consistent results than the MH algorithm for these diagnostics.

Zusammenfassung

In dieser Arbeit schätzen wir die A-posteriori-Verteilung der Eingabeparameter des HBV-SASK hydrologischen Modells mithilfe von Bayesscher Inferenz durch Markow-Ketten-Monte-Carlo (MCMC) Algorithmen ab. Dann wird das Modell mit dem Mittelwert oder dem Modus der A-posteriori-Verteilung der Parameter wahrscheinlich die beobachteten Abflussmengen nah simulieren. Wir beschreiben drei MCMC Algorithmen – den Metropolis-Hastings (MH) Algorithmus, Delayed Rejection Adaptive Metropolis Algorithmus (DRAM), und Differential Evolution Adaptive Metropolis (DREAM) Algorithmus – sowie zwei simple Parallelisierungen für die ersten beiden Algorithmen. Um diese Algorithmen mathematisch zu analysieren, erklären wir die Quadratwurzel des mittleren quadratischen Fehlers und den mittleren absoluten Fehler als Fehlerfunktionen, den Monte Carlo Standardfehler, effiziente Stichprobengröße und Autokorrelation, sowie die $\hat{R}$ Diagnostik zur Konvergenzanalyse. Wir werten die drei MCMC Algorithmen mit diesen Diagnostiken zusammen mit Spurdiagrammen, Histogrammen und weiteren Diagrammen aus. Wir bestimmen zuerst geeignete Parameterwerte für jeden Algorithmus nach diesen Diagnostiken, indem wir sie mit dem HBV-SASK hydrologischem Modell auf dem Banff Flussgebiet auswerten. Dann verwenden wir jeden MCMC Algorithmus mit diesen Parametern, um Parameter für das HBV-SASK Modell zu erhalten, die auf verschiedenen Trainingsdatensätzen trainiert wurden. Danach führen wir das HBV-SASK Modell mit diesen trainierten Parametern auf weiteren Evaluationsdatensätzen für die Banff und Oldman Flussgebiete aus.

Insgesamt verhalten sich alle Algorithmen gleich gut im Bezug auf die Fehlerfunktionen, wenn sie auf Datensätzen ähnlich der Trainingsdatensätze ausgewertet werden. Die Parameter, die wir durch den DREAM Algorithmus erhalten haben, verhalten sich schlechter als die der anderen beiden Algorithmen, wenn wir sie auf Evaluationsdatensätzen mit anderem Verhalten als dem jeweiligen Trainingsdatensatz verwenden. Allerdings erreicht DREAM deutlich bessere Ergebnisse für die effiziente Stichprobengröße und Autokorrelation sowie für die Konvergenzdiagnostic. Der DRAM Algorithmus weist bessere und konsistentere Ergebnisse als der MH Algorithmus für diese Diagnostiken auf.

Contents

1 Introduction

In many environments, it is unclear which input parameters to use to achieve the best results for a specific use case. Sometimes, only the upper and lower bounds of those input parameters are known. Therefore, additional methods must be used to estimate appropriate values for the input parameters based on the desired use case. One of these methods is to use Bayesian inference and estimate the parameter's posterior distribution through the Markov Chain Monte Carlo (MCMC) procedure.

An example of such an environment is the HBV-SASK model, a conceptual hydrological model developed at the University of Saskatchewan for educational purposes. This model estimates streamflow values, which can, for example, be used to predict floods. The appropriate parameter values change depending on the body of water the model evaluates. This hydrological model is the focus of this paper. We use Bayesian inference to estimate the posterior distribution of the model's parameters through MCMC algorithms using a training dataset to run the model. Then, the mean or mode of the parameters' posterior distribution will likely offer good results for other time spans outside the training dataset as well by closely simulating the observed streamflow values. [1]

We execute multiple MCMC algorithms – specifically, the basic Metropolis-Hastings (MH) algorithm [2, 3], the Delayed Rejection Adaptive Metropolis (DRAM) algorithm [4], and the Differential Evolution Adaptive Metropolis (DREAM) algorithm [5] – under different settings such as different likelihood functions, different covariance matrices of the proposal distribution, or burn-in lengths. We estimate the convergence rate, effective sample size (ESS), and Monte Carlo standard error (MCSE) for the resulting Markov chains. Additionally, we examine how well the HBV-SASK model runs with the inferred parameters on training and test data based on error functions like the root mean square error (RMSE) and mean absolute error (MAE). Some diagnostics, specifically about the convergence rate, require multiple Markov chains. Therefore, algorithms that normally use and create only one chain are run multiple times in parallel to obtain results for multiple chains. Based on these diagnostics, we determine good parameters for each MCMC algorithm. Afterwards, we perform additional runs with each MCMC algorithm on further training datasets to obtain multiple parameter sets. We then test those model parameters on different evaluation datasets.

For clarity and simplicity, whenever we mention *model* we mean the HBV-SASK model, and when we refer to an *algorithm*, we mean an MCMC algorithm, unless explicitly stated otherwise. From Chapter 4 onwards, *dimension* generally stands for the HBV-SASK model's parameters, as every sample during the algorithm executions represents a set of the model's parameters, and *parameter* for the MCMC algorithms' parameters.

Additionally, M refers to the total number of Markov chains used in an MCMC run, N to the number of samples per chain in an MCMC run, and d to the number of dimensions per sample, e.g., the number of parameters the model has. Furthermore, $z_{i,j}$ stands for the ith sample of the jth chain, and may be shortened to z_i if there is only a single chain.

In the remainder of this chapter, we introduce the HBV-SASK model, the training dataset, and the setup we use to run the algorithms. In the next chapter, we explain the basics of Bayesian Inference and MCMC algorithms. We describe the diagnostics we use to evaluate the results in Chapter 3. Chapters 4 to 6 describe the fundamentals and the evaluations of each algorithm. These chapters also contain the final parameter values we decided on for each algorithm. Chapter 7 contains model parameters trained on various training datasets with each algorithm as well as the comparison between the simulated and the measured streamflow values. Lastly, Chapter 8 concludes this thesis with a summary and ideas for potential follow-up works.

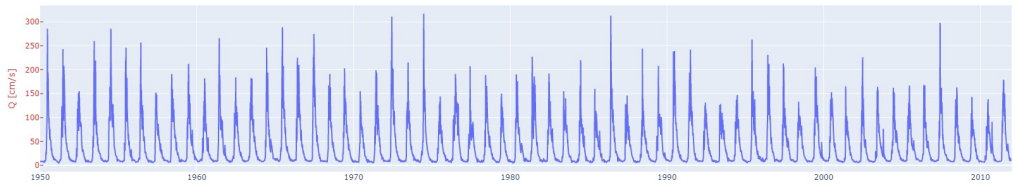
1.1 HBV-SASK Model

The common use case for the HBV-SASK model is to predict the daily streamflow for the Oldman River and the Bow River. Thus, the model already possesses data over the last years for these two rivers. The data is named and stored as *Oldman basin* and *Banff basin*, since the latter's data was measured at Banff. We use the latter nomenclature, as most papers do, which is also the nomenclature used within the model's code. The former basin contains data recorded over 30 years, from 1979 to 2008, whereas the latter has a total length of 62 years, from 1950 to 2011. The measured streamflow values of both basins are shown in Figure 1.1. There we see that the Banff basin's streamflow is far more periodic, whereas the Oldman basin has very low streamflow most of the time, but also a few peaks surpassing the streamflow values of the Banff basin. [1, 6]

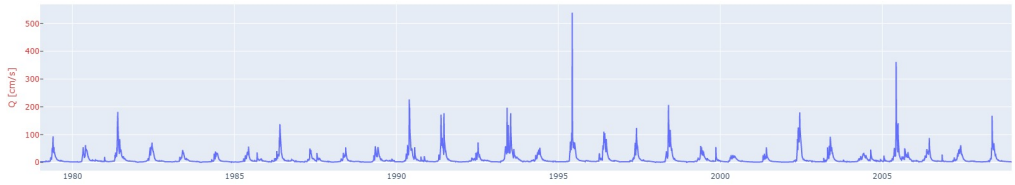
The HBV-SASK model possesses 10 input parameters to adjust and optimize the model to analyze the desired body of water. The 10 input parameters of the model, their lower and upper bounds, and their default values are listed in Table 1.1. [1, 6]

The *TT* parameter represents the air temperature threshold in °C for melting/freezing and to distinguish rain and snow, whereas *C0* defines the base melt factor in mm/°C per day. *ETF* is a parameter to correct temperature anomalies due to potential evapotranspiration. *LP* defines the limit for soil moisture below which evaporation becomes limited by supply and interacts with *FC*, which describes the maximum amount of water the soil can preserve. Lastly, *FRAC* defines the fraction of soil release that enters the fast reservoir. The other parameters have no concrete natural meanings, but are parameters needed for the calculation, such as shape parameters. [1, 6]

We use the first 20 recorded years of the Banff dataset, that is, from 1.1.1950 to 31.12.1969, with a spin-up length of 3 years as the training data while testing the parameters of the MCMC algorithms. This timespan is shown in Figure 1.2. The spin-up phase is needed so that the internal states, i.e., additional parameters of the model besides the 10 input parameters, can adjust themselves. We introduce more datasets in



(a) Banff basin



(b) Oldman basin

Figure 1.1: Measured values for both basins.

Table 1.1: The parameters of the HBV-SASK model.

Name	Lower bound	Upper bound	Default
TT	-4	4	0.0
C0	0	10	5.0
ETF	0	1	0.5
LP	0	1	0.5
FC	50	500	100
beta	1	3	2.0
FRAC	0.1	0.9	0.5
K1	0.05	1	0.5
alpha	1	3	2.0
K2	0	0.05	0.025

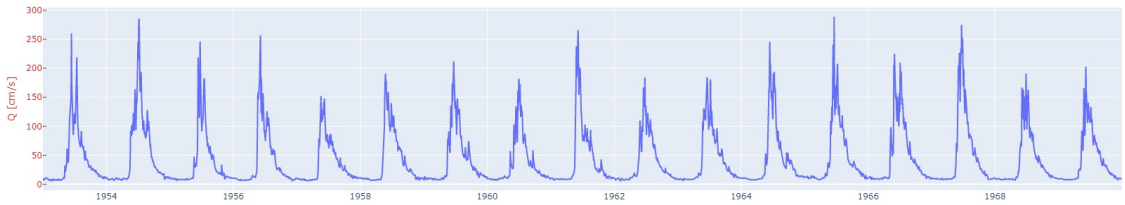


Figure 1.2: Measured data of the Banff basin used as training data.

Chapter 7, but this training dataset is the most important as we use it the most.

The model can return different types of outputs, such as error values over the entire run. We select the option to return the simulated and measured data for each day separately, so we can later make more distinguished decisions.

1.2 Hardware Specifications

Every algorithm is implemented in Python, and every test in this thesis is run on a desktop with an Intel(R) Core(TM) i7-4790 CPU @ 3.60 GHz, 3601 MHz, 4 cores and 8 logical processors, as well as 16GB of RAM using Windows 10 Home. With this hardware, a single execution of the model on our selected training set takes roughly 0.58s. This further increases if multiple chains execute it in parallel, with roughly 0.78s for 4 parallel executions and 1.23s for 8. Therefore, we limit ourselves to 10,000 iterations per chain for most executions, as more would be too time-consuming. Furthermore, we mostly use only four chains in our MCMC runs because there are only four cores, and for additional time saving.

1.3 Previous Works

The bachelor thesis of Zhou, Chengjie [7] heavily inspired this paper as it also works with the HBV-SASK model and MCMC algorithms; however, our paper has some significant differences from their paper. The main difference is that we utilize more diagnostics in all of our tests to determine good algorithm parameters. Where they used mostly just the execution time and error values to evaluate the performance, we additionally inspect the ESS, convergence rate, histograms, and sometimes the MCSE. We also use one additional algorithm, the DRAM algorithm, and leave out one algorithm that they explored, which had drastically worse execution times compared to other algorithms in their tests. Furthermore, they only used 7 model parameters, thus excluding the *LP*, *alpha*, and *K1* parameters, and used the Oldman basin in their training. Moreover, they used multiple univariate Gaussian distributions to propose new samples in certain algorithms, whereas we use a multivariate Gaussian distribution.

There are some more differences, such as the training set we use is with a period of 20 years longer than theirs with a period of 9 years, or that we generally perform more iterations per algorithm execution, with 10,000 per chain compared to 10,000 overall as they did. However, the previously described differences are sufficient to justify another thesis on this topic.

2 Basics of Bayesian Inference

The simplest way to estimate the distribution of parameters is to use only the observed output or measured data for the prediction. However, this can be problematic, for example, if the obtained data is small. To illustrate, if we predict the probability of a coin flip landing on heads and have observed only three coin tosses that each landed on heads so far, we would estimate a probability of 100% for the coin landing on heads. But assuming a fair coin, i.e., a coin where heads and tails have the same probability, the previous observation has a $(50\%)^3 = 12.5\%$ probability of occurring. Thus, if we include our knowledge or expectation into our estimation, we can make safer predictions and exclude impossible results beforehand. The calculation of the posterior distribution, the combination of observed data and knowledge, is described in Bayes' theorem. [2]

2.1 Bayes' Theorem

As mentioned before, we approximate the posterior distributions of the parameters according to Bayes' theorem to determine parameter values with which the model performs better:

$$\pi(z|y) = \frac{\pi(y|z)\pi(z)}{\pi(y)}, \tag{2.1}$$

where z represents the model's parameters and y the model output. Here, $\pi(y|z)$ is called the likelihood, $\pi(z)$ the prior, and $\pi(y)$ the evidence. The evidence is a constant in regard to z used to normalize the posterior so that it is an actual probability distribution. The prior represents the prior knowledge or expectation we have about the parameters. In our case with the HBV-SASK model, the prior is a uniform distribution ranging from each parameter's lower bound to its upper bound since we have no additional knowledge. The likelihood function represents how likely the model output is under the correct parameter values. In a perfect setting, the model output would replicate the ground truth values, i.e., the measured values. Consequently, the likelihood could simply be a distribution with high probability for the ground truth values and thus output values of the model, and probability zero for other values. But since the model and thus the output generally contain some inaccuracies, an error function is added to represent this. A common choice for such an error function is a Gaussian distribution centered on the ground truth values of the model, but other distributions are also possible options to represent this uncertainty. [2, 3, 8]

2.2 Rejection Sampling

Calculating the concrete posterior is generally difficult, so it is often approximated through numerical sampling. A rather simple and intuitive method to approximate a distribution $p(z)$ through sampling is rejection sampling. It uses similar ideas to the MCMC approaches, but is easier to understand, so we explain rejection sampling's basics first. Rejection sampling requires the target distribution to be known up to a constant, i.e., $jp(z)$ is evaluable for some fixed j, a proposal distribution $q(z)$, and a constant k such that $kq(z) \geq jp(z)$ holds for all values of z. The first of those requirements is fulfilled, for example, for a posterior when the likelihood and the prior are known, but the evidence is unknown. The last two requirements can be fulfilled by, e.g., selecting a Gaussian distribution as the proposal distribution and selecting a very large k. Then, in each iteration, the algorithm generates a new sample z' from the proposal distribution $q(z)$, evaluates the acceptance probability

$$\alpha = \frac{jp(z')}{kq(z')}, \tag{2.2}$$

and then accepts the new sample z' with probability α (e.g., by sampling a value u from a uniform distribution ranging from 0 to 1 and accepting z' if $\alpha > u$). After this is repeated multiple times, according to the Strong Law of Large Numbers, the sample mean will eventually be almost equal to the actual mean $\overline{m}$ since the samples are independent and identically distributed, that is

$$\lim_{N\to\infty} \frac{1}{N}(Z_1 + Z_2 + \cdots + Z_N) = \overline{m}. \tag{2.3}$$

As can be seen in Equation 2.2, the smaller the gap between $jp(z)$ and $kq(z)$ is, the larger α is, and the fewer samples are rejected. Therefore, it is beneficial if the proposal distribution closely mimics the behavior of the target distribution, such as having their minima and maxima for the same values. This makes the selection of the proposal distribution and k more difficult. [2, 3]

An illustrative example for this is given in Figure 2.1, where a scaled target distribution, a scaled proposal distribution, and the position of the new sample z' are shown. For the sampled z', α will be fairly small and z' will likely be rejected, which is the intended case as $jp(z')$ is small compared to $kq(z')$. In general, this proposal distribution is poorly chosen for the target distribution, as the area between the two distributions is large compared to the area for which they intersect. One can imagine that the value u sampled from the uniform distribution determines a point on the vertical line, with $u = 1$ setting the point at $kq(z')$. Thus, if that point lies below $jp(z')$, then the sample is accepted as it lies within the target distribution. [2]

The samples obtained through rejection sampling then represent the target distribution, after a sufficient number of iterations. Unfortunately, this method does not scale well in higher dimensions, and so other methods are used in high-dimensional problem cases, such as MCMC. [2, 3]

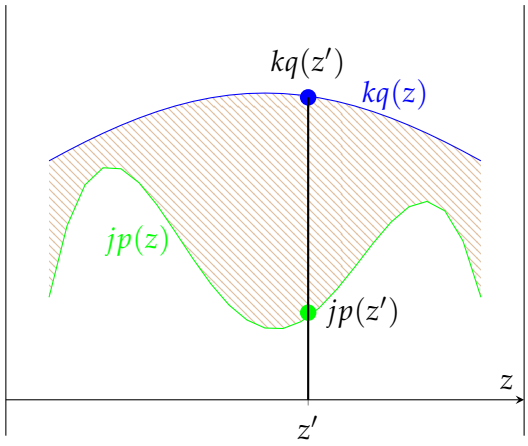


Figure 2.1: Example for rejection sampling, with the proposal distribution in blue and the target distribution in green

2.3 Markov Chain Monte Carlo

Before we describe the Metropolis-Hastings (MH) algorithm, a basic MCMC algorithm, we first discuss some necessary fundamentals of Markov chains.

2.3.1 Markov chains properties

First of all, the main trait of Markov chains, also called the **Markov property**, is that the current step depends only on the current state. Thus, we can define a transition probability $T_k(z_k, z_{k+1}) = p(z_{k+1} \mid z_k)$ that defines the probability of moving from z_k to z_{k+1} at the kth step regardless of previous states. For a **homogeneous Markov chain**, the transition probabilities depend only on the current and next state, and not on the current step number. [2, 3]

An **invariant**, or **stationary distribution** $p^*(z)$ of a Markov chain is a distribution that remains unchanged regardless of which transitions are performed, that is

$$p^*(z) = \sum_{z'} T(z, z')p^*(z). \tag{2.4}$$

Thus, if we can create a stationary distribution equal to the target distribution and reach it, we can sample from the target distribution. A sufficient condition for a distribution to be stationary is for it to be **reversible**, which is also called to fulfill **detailed balance**. This means that being at z and transitioning to z' has the same probability as being at z' and transitioning to z, i.e.,

$$p^*(z)T(z, z') = p^*(z')T(z', z). \tag{2.5}$$

Another property we require for our Markov chain is to be **ergodic**, that is, it possesses exactly one stationary distribution to which it eventually converges. Besides a few exceptions, homogeneous Markov chains are also ergodic. [2]

2.3.2 Metropolis-Hastings Algorithm

The basic MCMC algorithm is the Metropolis-Hastings (MH) algorithm, which most other MCMC algorithms are based on. The MH algorithm shares some similarities with rejection sampling as it requires the target distribution to only be known up to a constant and also uses a proposal distribution $q(z \mid z_k)$. But for the MH algorithm, the proposal distribution also depends on the current sample z_k. Thus, an initial sample must be manually selected as a starting point to draw the first sample from the proposal distribution. We can technically choose the proposal distribution freely. However, the proposal distribution affects the performance of the algorithm, as it changes which new samples z' are drawn from it. [2, 3]

The formula for the acceptance probability of the new sample z' is also different, namely

$$\alpha(z' \mid z_k) = \min\left\{1, \frac{p(z')q(z_k \mid z')}{p(z_k)q(z' \mid z_k)}\right\}, \tag{2.6}$$

where the min function is used so that $\alpha(z_k, z')$ is an actual probability, i.e., never larger than 1. For the case of a symmetric proposal distribution, that is, when $q(x \mid y) = q(y \mid x)$, the proposal distributions in Equation 2.6 cancel each other out. For the symmetric case it is thus apparent that if $p(z') \geq p(z_k)$, i.e., if z' is more likely than z_k for the target distribution, then z' will always be accepted. This is not necessarily the case with asymmetric proposal distributions. Another difference to rejection sampling is that if z' is rejected, then the last sample z_k is added again to the list of samples. Thus, samples with a high probability in the target distribution are contained more often in the final list of samples. Additionally, as a small side effect, the number of samples is the same as the number of iterations performed, which, for example, simplifies the examination of multiple independent runs. Algorithm 1 gives a pseudocode description of the MH algorithm. [2, 3]

The samples obtained with the MH algorithm form a Markov chain, as the algorithm's proposal distribution and acceptance probability depend only on the current sample. The transition probability from the current sample z_k to a new sample z' is the probability of sampling z' times the probability of accepting z', that is

$$T(z_k, z') = q(z' \mid z_k)\alpha(z' \mid z_k). \tag{2.7}$$

This is clearly homogeneous as it is independent of the number of iterations done before, so we already omitted the subscript for T. Therefore, we only require the MH algorithm to be reversible, and it will consequently possess only the target distribution as its invariant distribution. To prove that the MH algorithm is reversible, we show that Equation 2.5 holds with Equations 2.6 and 2.7 plugged in:

$$\begin{aligned} p(z_k)q(z' \mid z_k)\alpha(z' \mid z_k) &= p(z_k)q(z' \mid z_k)\min\left\{1, \frac{p(z')q(z_k \mid z')}{p(z_k)q(z' \mid z_k)}\right\} \\ &= \min\left\{p(z_k)q(z' \mid z_k), p(z')q(z_k \mid z')\right\} \\ &= p(z')q(z_k \mid z')\min\left\{\frac{p(z_k)q(z' \mid z_k)}{p(z')q(z_k \mid z')}, 1\right\} = p(z')q(z_k \mid z')\alpha(z_k \mid z'). \end{aligned} \tag{2.8}$$

Algorithm 1 Metropolis-Hastings Algorithm

Require: Target distribution $p(x)$, proposal distribution $q(x \mid y)$, initial sample z_0
Ensure: List of samples from the target distribution
1: Initialize list L as an empty list
2: **for** $k \leftarrow 0$ to N **do**
3: Sample z' from $q(x \mid z_k)$
4: Calculate $\alpha(z' \mid z_k) = \min\left\{1, \frac{p(z')q(z_k|z')}{p(z_k)q(z'|z_k)}\right\}$
5: Sample u from a uniform distribution ranging from 0 to 1
6: **if** $\alpha(z_k, z') > u$ **then**
7: $z_{k+1} \leftarrow z'$
8: **else**
9: $z_{k+1} \leftarrow z_k$
10: **end if**
11: Add z_{k+1} to L
12: **end for**
13: **return** L

If the new sample is not accepted, then $z_k = z_{k+1}$ and thus reversibility is trivially satisfied. [2, 3]

Consequently, the initial sample does not affect if the MH algorithm reaches the stationary distribution, but it affects how many iterations it requires to do so. The time it takes to reach the stationary distribution is called the burn-in phase. Since we only want to evaluate samples from the target distribution, samples obtained during the burn-in phase are of no value and are generally omitted before further evaluating the samples. The length of the burn-in phase can be examined through scatter plots or trace plots. A good scatter plot is one where the samples roughly recreate the target distribution, and a good trace plot is when it looks like a graph of white noise, i.e., when the samples are uncorrelated. [3]

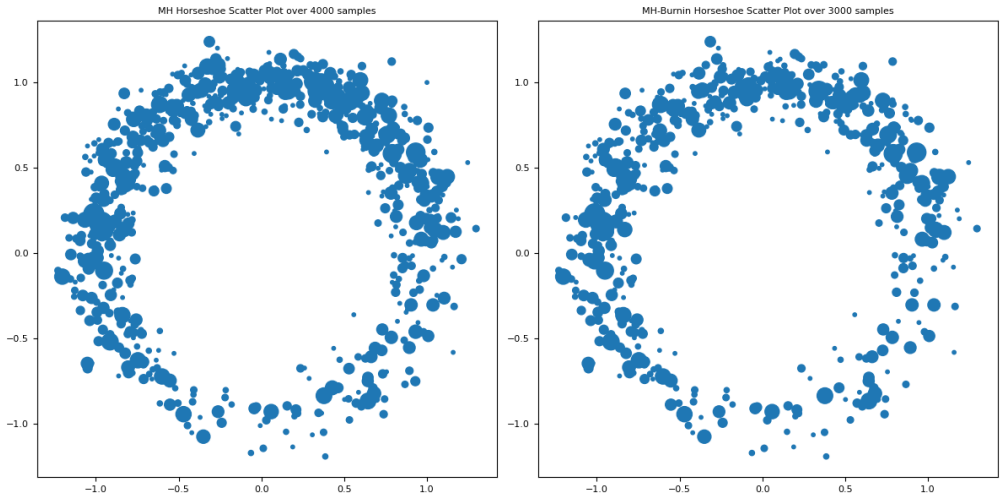
To illustrate this, we use the simple function

$$p(z)\alpha \exp -\frac{1}{0.02}(\sqrt{z_1^2 z_2^2} - 1)^2 - \frac{1}{2}(z_2 - 1)^2, \tag{2.9}$$

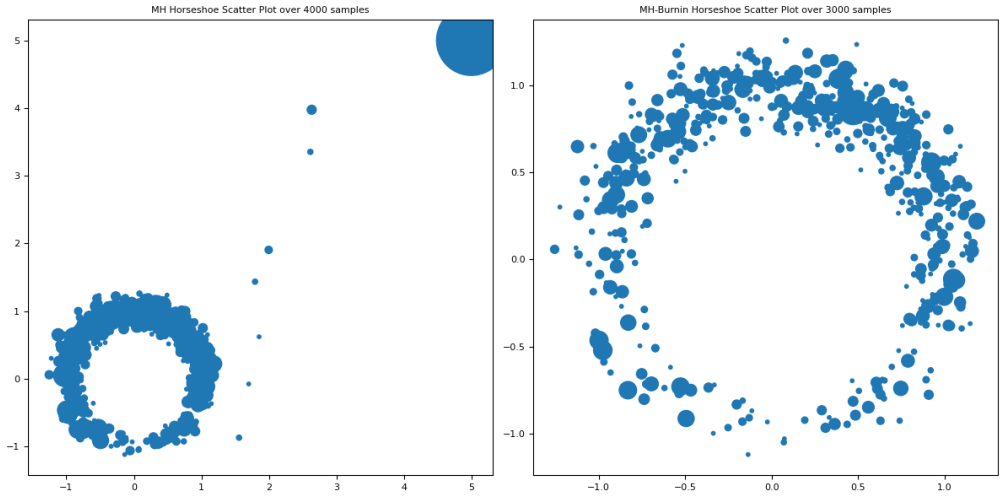
as target distribution where z_1 and z_2, as an exception, refer to the first and second dimension of a single sample and not the first and second sample in a chain, respectively. This simulates a circle with a gap at the bottom, which is why we call it the *horseshoe* function. [3]

We execute an MH run with 4000 iterations, starting at $(1, 1)$ and using a Gaussian proposal distribution with an identity matrix scaled down to 0.5 as the covariance matrix, which is the optimal covariance matrix for this function [3]. We show the scatter plot of the results in Figure 2.2a without and with burn-in of 1000 samples. More occurrences of the same value are represented by larger circles. In this figure, we see that the initial sample with value $(1, 1)$ is removed during the burn-in phase. But, except for this

change, there is no easily noticeable difference, which shows that the model quickly converges and that a shorter burn-in phase would suffice. If we select a starting sample further away from the horseshoe, e.g., at $(5, 5)$ like in Figure 2.2b, then there is a clear difference between with and without burn-in. We also plot the trace plots for both runs in Figure 2.3. In Figure 2.3b, we can see bad behavior in the beginning, where the samples stay at the starting point. But afterwards, and in the entirety of Figure 2.3a, we can see the white noise-like results we mentioned before, indicating low correlation.



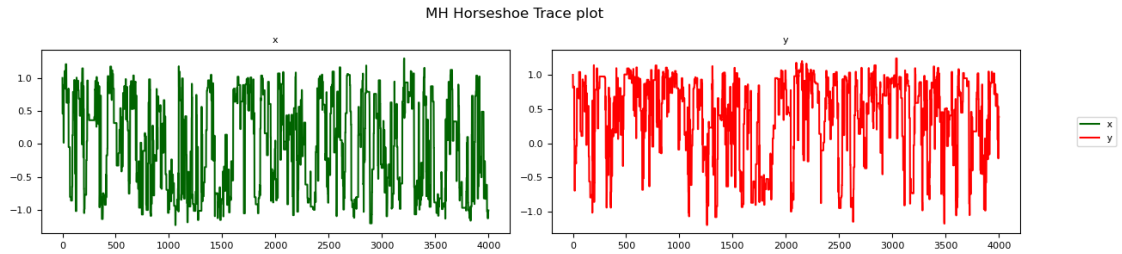
(a) Starting at $(1, 1)$



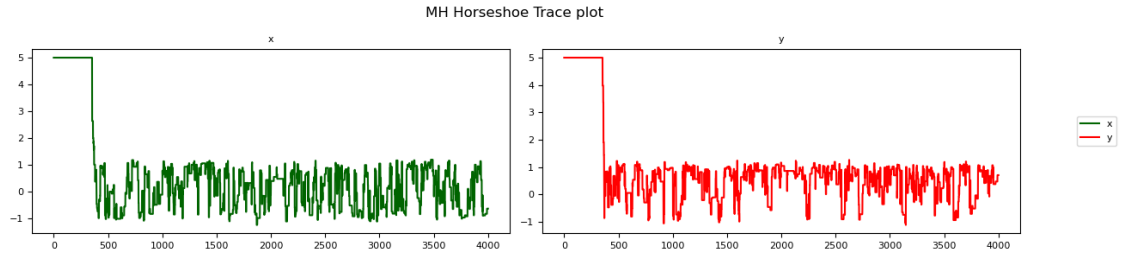
(b) Starting at $(5, 5)$

Figure 2.2: Scatter plots for MH runs on the horseshoe function without and with burn-in (1000 samples) with different starting samples.

So, the MH algorithm allows us to sample from higher-dimensional problems, unlike rejection sampling. But the samples are no longer independent of each other as the next



(a) Starting at $(1, 1)$



(b) Starting at $(5, 5)$

Figure 2.3: Trace plot of both dimensions for MH runs on the horseshoe function without burn-in with different starting samples.

step depends on the current sample, and thus, the Strong Law of Large Numbers is not applicable. To make the samples from the MH algorithm virtually independent, we can instead just examine every kth sample. The resulting sample size is called the **effective sample size**, and so we name k the **effective sampling step size**. The effective sampling step size for the samples in Figure 2.2a is 45 (and 44 with burn-in). We discuss, among others, multiple ways to estimate the effective sample size in the next chapter, specifically in Section 3.4. [3, 9]

3 Diagnostics

As we already mentioned in Chapter 2, the effectiveness of the MCMC algorithms to determine other models' parameters depends on their own parameters. For example, the proposal distribution affects how new samples are created, or the likelihood function influences the acceptance probability. Therefore, before we run the MCMC algorithms to obtain fitting parameters for the HBV-SASK model, we first need to test which parameters are appropriate for the MCMC algorithms so that they approximate the model's parameters well and quickly converge to the target distribution. We use certain diagnostics to evaluate how good the current algorithm parameters are, which we introduce in this chapter.

Since most of these diagnostics have no official name to differentiate them, we simply use the name of the first author of the paper from which we learned the respective diagnostic. Furthermore, as we have multiple diagnostics per category, we generally only show the results of one diagnostic per category unless there are surprising differences, since most methods per category have the same underlying idea and only have slight differences. The diagnoses all happen after the burn-in phase, so terms such as *total sample size* refer to the size after samples were removed due to burn-in.

3.1 Error functions

In the end, our main goal is to find parameters to make the model simulation as close as possible to the real measured data and minimize the error between them. To evaluate this error, we use the **mean absolute error** (MAE) and **root mean square error** (RMSE) functions. The former's definition is

$$\text{MAE} = \frac{1}{n}\sum_{j=1}^{n}|y_j - \hat{y}_j|, \tag{3.1}$$

whereas the latter's is

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{j=1}^{n}(y_j - \hat{y}_j)^2}, \tag{3.2}$$

where y_j is the simulated value and $\hat{y}_j$ the corresponding ground truth value. The RMSE function is more sensitive to outliers due to the square in the sum, whereas the MAE function is more stable, but consequently weighs small errors as strongly as large differences. For reference, using the default parameters as in Table 1.1 on the training set results in an MAE of 39.42 and an RMSE of 74.10. Thus, we use both functions: For

each algorithm run, we determine the sample that appears the most (i.e., the mode) and the mean of the samples. Using these as the model's parameters, we perform two more model runs of the training set and evaluate the results of each with the MAE and RMSE functions. We name these results mode MAE, mode RMSE, mean MAE, and mean RMSE, respectively. Of those two parameter values, the mean is generally more relevant than the mode since the former represents every sample, whereas the latter can be an outlier. [1, 7, 10]

We use another dataset as a test dataset to evaluate the error functions. The test dataset contains the next 10 years after the training dataset, i.e., from 1970 to 1979. However, these test error values generally show the same behavior as the training error values, with the test errors being slightly larger. Furthermore, the error values for some results are smaller than when using the training dataset again. This suggests that these better test results are not good results stemming from good parameter settings, but just coincidentally fit the test data better than the training data. This is unwanted, since it is a sign that the current parameters can not fit the data. Thus, we have decided to mostly rely on the training set for the error values when comparing the parameters of the MCMC algorithms. If we discuss the error values from using the test set, we specifically mark them as such. So, for example, only mean MAE refers to the MAE value obtained from using the mean of samples to do a model run over the training set, whereas test mean MAE refers to the MAE value obtained from using the mean to perform a model run over the test set.

3.2 Monte Carlo Standard Error

A common tool in statistics is to examine the standard error, that is, the standard deviation σ over all samples. The true σ is generally unknown, but we can estimate it with the sample mean $\bar{z}_j$ of the samples from chain j using the common variance formula. This estimation of σ is called the **Monte Carlo Standard Error** (MCSE) $\hat{\sigma}_j$ and is calculated as

$$\hat{\sigma}_j = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(z_{i,j} - \bar{z}_j)^2}. \tag{3.3}$$

Thus, $\hat{\sigma}_j$ is the approximated standard error for chain j. [11, 12]

Another way to estimate the MCSE is to use smaller batches first, instead of the entire sample size at once. For this, we first divide the samples into K non-overlapping batches of length b. If b does not divide the sample size evenly, then we ignore the last samples that do not fit into a full batch. Then we can basically use Equation 3.3 again, but with the mean over the batches $\bar{z}_{k,j}$ per chain j instead of $z_{i,j}$, that is

$$\hat{\sigma}_{batch,j} = \sqrt{\frac{b}{K}\sum_{k=1}^{K}(\bar{z}_{kj} - \bar{z}_j)^2}. \tag{3.4}$$

We use a batch length of 500 when calculating the batch MCSE. The advantage of batch MCSE is that we can also understand the time progression more clearly, whereas MCSE always returns the same values for the same samples, regardless of how the samples are ordered. For example, if half of all samples are the same value, then the MCSE result will be small. However, the batch MCSE differs if those samples with the same value are all consecutive and thus in the same batches, or if those samples are evenly spread so that every batch consists mostly of this value. [12]

The MCSE is evaluated per chain and per dimension. Therefore, we examine the mean over the chains and dimensions to reduce the number of different MCSE results. However, how to evaluate the MCSE results is a bit more complicated: In other fields, the lower the variance, the better the result. But for sampling as in our case, low variance *can* also indicate that the algorithm is stuck and does not explore the whole parameter space. So, from MCSE alone, we can not evaluate if a high or low variance represents bad or good results, which we also see in the autocorrelation diagnostic, where bad results have low variance. Thus, we have to consider other diagnostics as well.

3.3 R-Convergence

In this section, we cover some methods to estimate if the current MCMC algorithm has converged to the target distribution. The fundamental idea is that if multiple chains that started at different positions have low variance between each other, i.e., each chain exhibits a similar behavior, then they likely converged to the same area. This area should be the target distribution because the MCMC algorithms converge only to the target distribution, as we showed in Section 2.3.2. If furthermore the variance within each chain is still high, it makes it more likely that they have actually converged to an area of high probability, and are not just stuck at a certain value. As the value to measure the convergence is generally abbreviated as $\hat{R}$, we name this diagnostic **R-convergence** to clarify which kind of convergence diagnostic we refer to. This convergence analysis requires at least two chains of samples, ideally with different starting points, to make a reliable and unbiased estimate. When the resulting value is close to 1, the method predicts that the algorithm has converged. [12]

Unfortunately, such a result is not a guarantee for convergence, and might be tricked by pseudo-convergence, e.g., when not all modes have been discovered yet. However, the only known alternative is to make a longer algorithm run by performing more iterations. This is not feasible in our case, where we try to do many runs to test different settings. Thus, these convergence diagnostics are the only real available option for us. Additionally, a large $\hat{R}$ value still indicates that the chains have *not* converged, so it still serves a purpose. We cover three methods to calculate this diagnostic: the traditional $\hat{R}$ according to Gelman and Rubin [13], as well as two improvements to that method. [12, 14]

3.3.1 Traditional $\hat{R}$

For this method, we first calculate the within-sequence variance W and between-sequence variance B. To calculate those values, we define the terms for the chain mean $\bar{z}_{\cdot j}$, the overall mean $\bar{z}_{\cdot\cdot}$, and the chain sample variance s_j^2:

$$\begin{aligned} \bar{z}_{\cdot j} &= \frac{1}{N}\sum_{i=1}^{N} z_{i,j}, \qquad \bar{z}_{\cdot\cdot} = \frac{1}{M}\sum_{j=1}^{M} \bar{z}_{\cdot j}, \\ s_j^2 &= \frac{1}{N-1}\sum_{i=1}^{N}(z_{i,j} - \bar{z}_{\cdot j})^2. \end{aligned} \tag{3.5}$$

The chain sample variance s_j^2 is almost the same as the MCSE, but is scaled slightly differently. This is the common definition of these values, to which we adhere. With those definitions, we can calculate W and B as

$$\begin{aligned} W &= \frac{1}{M}\sum_{j=1}^{M} s_j^2, \\ B &= \frac{N}{M-1}\sum_{j=1}^{M}(\bar{z}_{\cdot j} - \bar{z}_{\cdot\cdot})^2. \end{aligned} \tag{3.6}$$

From there, we can calculate an overestimate of the posterior's variance as

$$\widehat{var} = \frac{N-1}{N}W + \frac{1}{N}B, \tag{3.7}$$

and finally the $\hat{R}$ value as

$$\hat{R} = \sqrt{\frac{\widehat{var}}{W}}. \tag{3.8}$$

Originally, an $\hat{R}$ value of 1.1 was assumed to be sufficient to assume convergence. But due to the problem of pseudo-convergence, the lower the $\hat{R}$ value is, the more reliable it is, so only values of 1.01 and below are now treated as signs of convergence. [9, 13, 14]

3.3.2 Other $\hat{R}$ Methods

The first, simple improvement to the traditional $\hat{R}$ diagnostic is to split each chain in half and then do the same calculations with the increased number of chains. We thus name the resulting value **split-$\hat{R}$**. This way, the chains must already show similar behaviors in the beginning as in the end, making it harsher and thus more reliable.

The second improvement adds two more changes to *split-$\hat{R}$*, as it calculates two *split-$\hat{R}$* values as in the equations in Section 3.3.1: one with rank-normalized sample values $n_{i,j}$, and one with folded rank normalized sample values $f_{i,j}$. The final result is the larger of the two *split-$\hat{R}$* values, which is why we call it **rank/fold-$\hat{R}$**. [9]

Rank-normalization proceeds as follows: We first reduce each sample to a scalar value, e.g., by taking the sum over its dimensions. Then, we can assign a rank to each sample

from smallest to largest, with duplicates attaining their average rank. For example, *[0, 7, 29, 7]* is ranked as *[1, 2.5, 4, 2.5]*. Those ranks are then normalized as

$$n_{i,j} = \Phi^{-1}\left(\frac{z_{i,j} - 3/8}{NM + 1/4}\right). \tag{3.9}$$

For the folded rank-normalized values, the absolute difference from the median over all samples is first calculated before they are rank-normalized, i.e., the values $a_{i,j}$,

$$a_{i,j} = |z_{i,j} - \text{median}(z)|, \tag{3.10}$$

are ranked and normalized as in Equation 3.9 to obtain $f_{i,j}$. [9]

Since *rank/fold-*$\hat{R}$ is the most developed diagnostic of the three we described, we will mostly rely on it, unless one of the other convergence diagnostics shows noteworthy results. Since the other two diagnostics return a result per dimension, we take their mean, minimum, and maximum for the final evaluation.

3.4 Autocorrelation and Effective Sample Size

As mentioned in the previous chapter, since the samples from MCMC algorithms are not independent, we want to leave some samples out to make the remaining samples virtually independent. We can calculate the **effective sample size** (ESS) from the **effective sampling step size** (ESSS) by dividing the total sample size by the ESSS, and vice versa. Thus, the ESS describes the total number of virtually independent samples, and the ESSS describes how many samples should be left out between two samples to obtain the ESS. However, to calculate the ESSS and consequently the ESS, we require the autocorrelation values of the samples. So, we first describe how we calculate those values. [3, 9]

3.4.1 Autocorrelation

To calculate the autocorrelation for a single chain at lag k, we use the term

$$\hat{c}_k = \frac{c_k}{c_0}, \quad k = 0, \ldots, N-1, \tag{3.11}$$

where c_k is defined as

$$c_k = \frac{1}{n}\sum_{j=1}^{N-k} \hat{z}_{j+k}\hat{z}_j, \quad k = 0, \ldots, N-1, \tag{3.12}$$

and where $\hat{z}_k$ is z_k after being normalized to have zero mean through the sample mean. Therefore, even if all z_k are the same value, the $\hat{c}_k$ strictly decreases with larger k as fewer values are summed up. However, if the z_k differ, the autocorrelation decreases faster since the first sample is generally further from the mean value and thus larger

after normalization than subsequent samples. It is arguable if one should divide c_k by n or by $n-k$, but the latter would make the c_k even noisier. Therefore, we divide by n. We refer to the values c_k as autocovariance, and the normalized values $\hat{c}_k$ as **autocorrelation**, although some papers use those terms interchangeably. [3, 12]

We found two methods to calculate the autocorrelation for multiple chains. The first is from Vehtari, et al. [9]. The method begins by calculating the autocorrelation $\hat{\rho}_{k,m}$ per chain m for lag k like in Equation 3.11. It then calculates the overall autocorrelation $\hat{\rho}_k$ at lag k as

$$\hat{\rho}_k = 1 - \frac{W - \frac{1}{M}\sum_{m=1}^{M} s_m^2 \hat{\rho}_{k,m}}{\widehat{var}} \tag{3.13}$$

where s_m^2, W, and $\widehat{var}$ are calculated as in Equations 3.5, 3.6, and 3.7, respectively.

The second method is from Gelman, et al. [14]. It calculates the autocorrelation by first calculating the variogram V_k for each lag k as

$$V_k = \frac{1}{M(N-k)} \sum_{j=1}^{M} \sum_{i=k+1}^{N} (z_{i,j} - z_{i-k,j})^2, \tag{3.14}$$

and then calculates

$$\hat{\rho}_k = 1 - \frac{V_k}{\widehat{var}} \tag{3.15}$$

where $\widehat{var}$ is defined in Equation 3.7. [14]

3.4.2 ESS

ESS for a Single Chain

For the single chain case, we found only the method to select an ESSS from Bui-Thanh [3], which we thus call **esss_buithanh**:
If $\hat{c}_k$ is below a certain threshold in every dimension, then we can use every k-th sample to calculate the sample mean, as those samples are then virtually independent of each other. By selecting the threshold, we can decide how strict we want to be in determining the ESSS; the closer to zero, the more selective. Bui-Thanh uses a value of around 0.05 in their paper. However, we use the larger value of 0.5 since our problem case is more complicated than theirs, and it is thus more difficult to obtain low autocorrelation values. With this requirement, the worst possible ESSS is half of the total sample size of a chain, which, for example, occurs if all samples of a chain are the same value, i.e., are fully correlated. [3]

The final *esss_buithanh* value for a chain is the lag at which the autocorrelation is below the threshold for all dimensions, so the dimension with the highest correlation is the decisive result. For example, if all dimensions but *TT* have an autocorrelation value below 0.5 at lag 1, but *TT* only reaches this threshold at lag 100, then the *esss_buithanh* is 100 for this chain. [3]

ESS for Multiple Chains

For results from multiple chains, we have three methods to calculate the ESS: **ess_vehtari**, **bulk-ESS**, and **ess_gelman**.

The ESS calculation according to Vehtari, et al. [9] begins by calculating the autocorrelation values over all chains as in Equation 3.13 and then adds every two successive values together, that is

$$\hat{P}_k = \hat{\rho}_{2k} + \hat{\rho}_{2k+1}. \tag{3.16}$$

This way, $\hat{P}_k$ is strictly decreasing. By keeping only values before the first nonpositive $\hat{P}_t$, i.e., by keeping $\hat{P}_k \geq 0$ with k in $0, \dots t-1$, we remove unrealistic, negative values since we know that the ESSS can not be negative. These remaining $\hat{P}_k$ are then summed up to calculate the ESSS

$$\hat{\tau} = -1 + 2\sum_{i=0}^{t-1} \hat{P}_i, \tag{3.17}$$

and the ESS is the total sample size (over all chains) divided by $\hat{\tau}$. We calculate Equation 3.17 independently per dimension, e.g., if $\hat{P}_3 < 0$ for one dimension, then we sum up only the first three $\hat{P}_k$ for this dimension. On the other hand, if $\hat{P}_k \geq 0$ for k in $0, \dots, 100$ for another dimension, we sum up at least the first 100 values for the latter dimension. If we did not treat the dimensions separately, the $\hat{\tau}$ values of the first dimension could become negative as we would add almost 100 values, which are close to 0 or negative. If $\hat{P}_k$ contains no negative values in one dimension, then we abort the calculation of this diagnostic for all dimensions. Otherwise, the resulting ESS might be too loose since more samples would generally still result in more positive $\hat{P}_k$ values, and thus increase the ESSS and decrease the ESS. We signify this by returning *Not a Number* (NaN). [9, 12, 15]

Bulk-ESS functions almost the same as *ess_vehtari*, but it uses rank-normalized values as in Equation 3.9 to calculate the autocorrelation values over all chains. [9]

Our third method, *ess_gelman* from Gelman, et al. [14] is essentially the same as the first method for multiple chains, but it uses the autocorrelation values from Equation 3.15 instead. [14]

Usage

After calculating the different ESS values and their corresponding ESSS (or vice versa for *esss_buithanh*), we do an additional run of the diagnostics (except for autocorrelation and ESS). For this run, we reduce the sample size based on the largest ESSS according to the different methods, i.e., the harshest result for the ESS – as long as the resulting ESS value is not too small. We have decided to require at least 8 samples initially as ESS per chain, since some diagnostics do not work with too few samples, such as *split-*$\hat{R}$, and the worst achievable ESS is 2 even if samples are fully correlated because of *esss_buithanh*.

Since *ess_vehtari* and *ess_gelman* both return one value per sample dimension, we reduce those to a single value for this by taking their minimum. This represents the stricter, worst-case result.

For the single chain case, we naturally only use *esss_buithanh* as the other methods can not be calculated, and take the maximum ESSS over all dimensions as the final ESSS. With multiple chains, we use the three multi-chain methods as well as the maximum of *esss_buithanh* over all chains.

We have decided not to take the *esss_buithanh* values of each chain and apply them independently to each chain, since *esss_buithanh* is already very lenient due to the high minimum value of 0.5 we have selected. This also makes the comparison of results easier as we have only one max *esss_buithanh* value instead of one per chain.

We additionally record the mean and max (mean and min for *esss_buithanh*) of all diagnostics except for bulk-ESS, as it is a scalar. For *ess_vehtari* and *ess_gelman*, this represents the mean and minimum over the dimensions, and for *esss_buithanh*, the mean and maximum over the maximum ESSS per chain. However, the max (min for *esss_buithanh*) is the least important value of the three since it gives the best, most optimistic result.

3.5 Additional Diagnostics

Table 3.1: The diagnostics summarized.

Error functions	**MCSE**	**R-Convergence**	**ESS**
MAE	MCSE	(Traditional) $\hat{R}$	*esss_buithanh*
RMSE	batch MCSE	*split-*$\hat{R}$	*ess_vehtari*
		rank/fold-$\hat{R}$	Bulk-ESS
			ess_gelman

Table 3.1 summarizes the aforementioned diagnostics again. In addition to the more complicated diagnostics above, we also use simpler diagnostics that do not require complex explanations. Those include the measurements of the execution time of the algorithm, the number of unique samples obtained, and the overall acceptance rate of proposed samples.

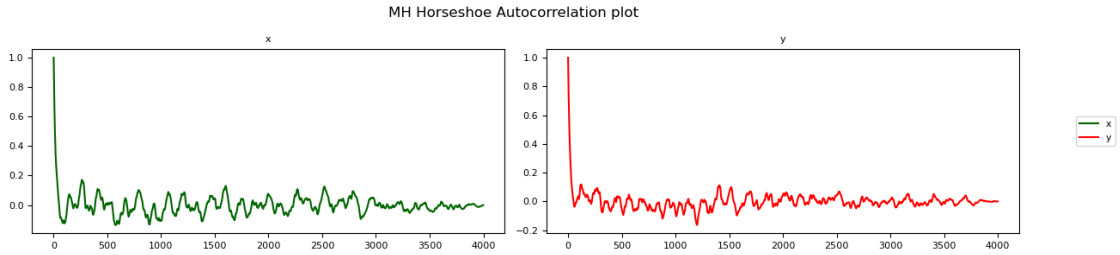
Most of these diagnostics are also represented in fitting diagrams, such as bar plots for the acceptance rates, boxplots, histograms and kernel density estimates (KDE) plots over the samples to roughly represent the obtained distribution, trace plots (time-series plots but over different data instead of time) for the autocorrelation values, and more. The KDE is an estimate of the distributions based on the samples, represented as a black line in the histograms. Only scalar values, for which we found no meaningful graphical representation, such as the execution time, are not stored in diagrams. Due to the resulting abundance of diagrams, we only show those that exhibit noteworthy results in this paper. [7]

3.5.1 Example

Using the horseshoe function from the previous chapter as an example again, we create plots to showcase some of the aforementioned diagrams in Figure 3.1.

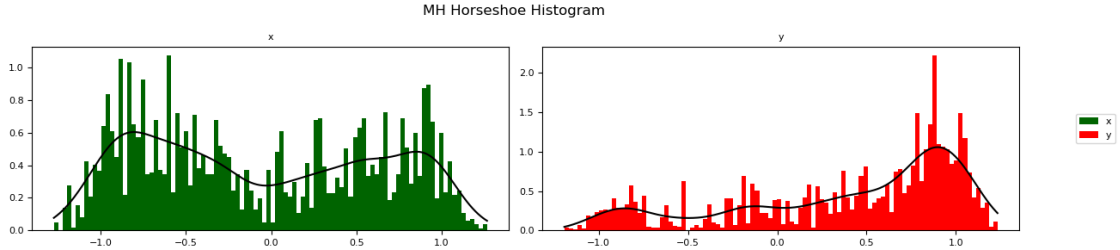
In the autocorrelation plots in Figure 3.1a, we can see that it quickly reaches low values and then fluctuates around 0, which is good behavior to show uncorrelated samples and thus results in an *esss_buithanh* of 45. The trace plots in Figure 3.1b also show this as the samples vary significantly over a large area.

The histograms and boxplots in Figures 3.1c and 3.1d show that the samples have the expected behavior: The samples for the x dimension are centered at 0 and symmetric, whereas they are biased towards positive values for the y dimension since the horseshoe is symmetric to the y-axis but not to the x-axis due to the gap at its bottom.

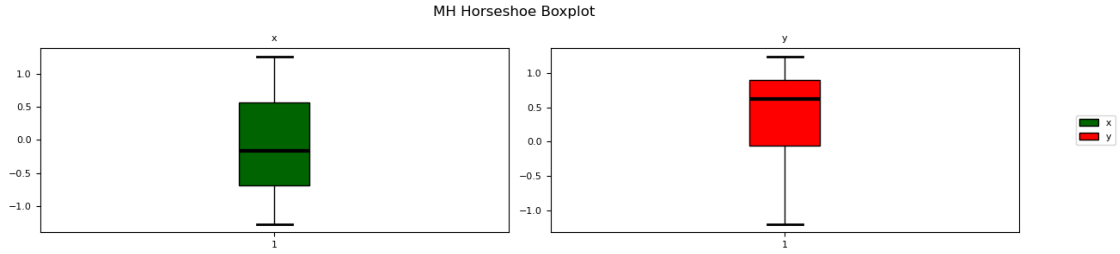


(a) Autocorrelation

(b) Trace plot



(c) Histogram



(d) Boxplot

Figure 3.1: Diagrams for the horseshoe example.

4 Metropolis-Hastings algorithm

With the preparations done in the previous chapters, we can now begin discussing our concrete test cases and their results. We start with the simple MH algorithm already described in Section 2.3.2. But first, we modify and simplify the general description by adjusting it to our concrete use case with the HBV-SASK model.

4.1 MH Algorithm with HBV-SASK

If we select the posterior $\pi(z|y)$ of our parameters as the target distribution $p(z)$ and plug Equation 2.1, the definition of the posterior, into Equation 2.6, the equation for the acceptance probability, we obtain

$$\begin{aligned} \alpha(z' \mid z_k) &= \min\left\{1, \frac{\pi(z'|y)q(z_k \mid z')}{\pi(z_k|y)q(z' \mid z_k)}\right\} = \min\left\{1, \frac{\pi(y|z')\pi(z')q(z_k \mid z')}{\pi(y|z_k)\pi(z_k)q(z' \mid z_k)}\right\} \\ &\left(= \min\left\{1, \frac{\pi(y|z')q(z_k \mid z')}{\pi(y|z_k)q(z' \mid z_k)}\right\} = \min\left\{1, \frac{\pi(y|z')}{\pi(y|z_k)}\right\}\right). \end{aligned} \tag{4.1}$$

As the evidence is constant, it cancels itself out in the second step. The first term in parentheses holds when z' and z_k are both within the parameters' boundaries and thus have the same prior probability due to the uniform prior we use. The last term in parentheses holds if, additionally, the proposal distribution is symmetric. We have to ensure that the sampled parameters are within the parameters' boundaries anyway, as the model might encounter errors otherwise, and we also use only Gaussian proposal distributions, which are symmetric. Therefore, we can directly work with the simplified acceptance probability formula

$$\alpha(z' \mid z_k) = \min\left\{1, \frac{\pi(y|z')}{\pi(y|z_k)}\right\}, \tag{4.2}$$

or specifically, its logarithmic version to avoid numerical instability

$$\log \alpha(z' \mid z_k) = \min\left\{0, \log \pi(y|z') - \log \pi(y|z_k)\right\}. \tag{4.3}$$

Thus, the acceptance probability is technically independent of the proposal distribution, although the proposal distribution still selects z'. Explicitly, the proposal distribution uses the last accepted sample as the mean of its Gaussian distribution to sample z'.

In summary, the proposal distribution creates a new parameter sample with which we run the model. With the resulting prediction and measured data – one pair per evaluated day – the algorithm calculates the likelihood of the current sample. The

algorithm compares the likelihood of the current sample to the likelihood of the last accepted sample to calculate the current sample's acceptance probability.

This means that we also need to execute the HBV-SASK model during each step of the MH algorithm. Thus, we require a set of data to execute the MCMC algorithm, which is the aforementioned training dataset. At the end, the obtained samples give an estimate of the target distribution's posterior and thus the model's parameters. Those parameter estimates could, for example, be the posterior's mean, mode, or median.

Additionally, since we obtain one prediction and measured data per day for which the model is run, we have to handle the large amount of data while calculating the likelihood. If we calculate the likelihood per day and then multiply those probabilities to obtain a single overall likelihood for a sample, then that probability would be very small and, more importantly, very volatile. As a simple example, if our training dataset spans one year, i.e., 365 days, our current sample has a likelihood of 0.49 per day, and the last accepted sample has a likelihood of 0.5 per day, then the acceptance probability of the current sample according to Equation 4.2 is

$$\min\left\{1, \frac{0.49^{365}}{0.5^{365}}\right\} \approx 0.000627. \tag{4.4}$$

Thus, even though each likelihood is fairly high and very close to the likelihood of the previous sample, the acceptance probability is still very small. Consequently, newly accepted samples rarely result in worse simulations than already accepted samples, because slightly worse simulations drastically decrease the likelihood. While this might sound good, it would be best if the samples varied slightly, e.g., in case of multiple peaks in the target distribution. We cover our ideas for different likelihood functions to handle this problem in Section 4.4.1.

To avoid numerical errors when comparing the acceptance probability to the value drawn from the uniform distribution ranging from 0 to 1, we work in the log-space. Therefore, we use Equation 4.3 to calculate the log acceptance probability and compare it to the log value of the value drawn from the uniform distribution. Since we work entirely with log probabilities, we omit to mention log every time for convenience, and explicitly mention when we are not talking about log probabilities.

4.2 Parallel MH

Since the basic MH algorithm works only on a single chain, we can not utilize every diagnostic we introduced, especially those regarding R-convergence. For this reason, we also have a parallel version of the MH algorithm where M different executions of the MH algorithm are run independently with the same parameters, besides optionally different starting samples for each chain. The results from those M different runs are then pooled together and treated as a single result like any result from a multi-chain algorithm. It is important to note that by doing this, each independent chain has to go through the burn-in phase to reach the target distribution. Thus, if we, for example, do

a parallel MH run with 2 chains and 5,000 iterations each for a total of 10,000 samples and a burn-in length of 2,000, we obtain only 6,000 usable samples. On the other hand, with a run of a normal MH algorithm with 10,000 samples and the same burn-in length of 2,000, we obtain 8,000 usable samples. So, one can not simply obtain NM usable samples by executing M runs with length N each. But, in our case, using M chains in parallel increases the running time by less than a factor of M, so that the parallel version obtains more usable samples in the same time than the basic version. Thus, we initially use the parallel MH version, and then compare the basic MH algorithm with our parallelized version after determining appropriate parameter values.

4.3 Initial Settings

We fix the parameters of the MH algorithm initially to the values shown in Table 4.1 and explain their meaning in the respective subsection. We do not list the likelihood function and proposal covariance matrix in this table because they are the first parameters we test. After finding the best values for those two parameters, we use those while testing other values for the parameters in Table 4.1. Otherwise, it would take too much time to test every likelihood function and proposal covariance matrix with multiple different parameter values. Therefore, we start with the parameters we think have the most impact on the results or for which we have the least knowledge of good values, or both, and move on with the ones we are more certain of.

Table 4.1: The other starting parameters of the MH algorithm.

Parameter Name	Value
Likelihood Scale	0.5
Boundary Handling	Fold
Starting Sample	Spread
Number of Chains	4
Burn-in Length	20%

4.4 Likelihood and Proposal Covariance Matrix

As mentioned, we start with the most important parameters for this algorithm, the likelihood function and the covariance matrix that the proposal distribution uses. These can hardly be treated independently, since they both heavily affect the acceptance rate, e.g., how bold the proposals should be and how accepting the likelihood function should be.

However, before we discuss the results, we first introduce the different likelihood functions and proposal covariance matrices we use.

4.4.1 Likelihood Functions

The problem described in Equation 4.4 is even worse in our case since a single HBV-SASK model run returns one result per day over the entire time frame that the model evaluates, which ends up at around 6000 data pairs of simulated and observed values. Thus, a simple likelihood function that calculates a likelihood per data pair and then combines them into a single likelihood via multiplication might not be the best approach. To test this, one of our likelihood functions does exactly that: It uses the *norm* function from the Python module *scipy.stats* to calculate the Gaussian log probability as in Equation 4.5. This function uses a constant scale value σ, the measured value as the mean $\hat{y}_j$, and the simulated value as y_j, i.e., the value we want to evaluate the probability density function for. The probabilities are then summed up (because we are in the log space) to obtain a scalar probability, calculating the same as Equation 4.6 where K is the number of data pairs. We thus call this likelihood function **scipy_log_gaussian**. While this function has the aforementioned problem of obtaining mostly very low probabilities and thus low acceptance rates, it also means that it is virtually impossible to accept a proposal that results in worse simulations than the current sample. Therefore, it essentially minimizes the error values and quickly moves towards areas with high probability densities, which can be seen as an advantage.

$$-0.5 \log(2\pi\sigma^2) - \frac{(y_j - \hat{y}_j)^2}{2\sigma^2} \tag{4.5}$$

$$-0.5 \log(2\pi\sigma^2)K - \sum_{i=1}^{K} \frac{(y_j - \hat{y}_j)^2}{2\sigma^2} \tag{4.6}$$

Based on this standard approach, we designed a few derivative versions that try to solve the problem shown in Equation 4.4. One idea is to take the mean of the probabilities instead of summing them up, which we thus call **scipy_post_mean_log_gaussian**. Notably, this is equivalent to taking the mean squared error (the RMSE without taking the square root) of the measured and simulated values first before scaling them by σ^2 and using them as the exponent of the Gaussian probability density function.

Similarly, instead of taking the mean afterwards, we can take the mean of the measured and simulated values, respectively, and then use the *norm* method to directly obtain a scalar probability. Therefore, we name this method **scipy_pre_mean_log_gaussian**.

Another idea is to take batches of measured and simulated values, calculate their mean, and then calculate the likelihood per batch. The special case of batches of size 1 is equivalent to *scipy_log_gaussian*, and the case of one single batch with full size is *scipy_pre_mean_log_gaussian*. In addition to those special cases, we use batches of size 30 and 365, which we name **scipy_monthly_mean_log_gaussian** and **scipy_yearly_mean_log_gaussian**, respectively.

One more proposition we came up with is to use 12 batches to capture potential monthly recurring behavior in the data. For simplicity, we assumed that every month has 30 days. Thus, one batch contains 30 consecutive data values and excludes the next

330 data values repeatedly (i.e., one batch contains the first 30 data points and the 361st to 390th data point, ...). Then we calculate the probability per batch and combine them into a scalar probability as in the other methods. We name this likelihood function **scipy_per_month_log_gaussian**.

The last method is different as it does not only rely on the Gaussian distribution. For this likelihood function, we assume that the error residuals $e_k(\mathbf{z})$, i.e., the difference between the measured and simulated values, represent an autoregressive model of order 1 (AR(1)). This means that the current value depends linearly on the previous and some white noise η_k

$$e_k(\mathbf{z}) = \phi e_{k-1}(\mathbf{z}) + \eta_k. \tag{4.7}$$

Combining this with Equation 4.6, we obtain as the likelihood function

$$\begin{aligned} &-\frac{K}{2}\log(2\pi) + \frac{1}{2}\log(1-\phi^2) - \frac{1}{2}(1-\phi^2)\hat{\sigma}^2 e_1(\mathbf{z})^2 \\ &\quad - \sum_{k=2}^{K}\log\hat{\sigma}^2 - \frac{1}{2}\sum_{k=2}^{K}\left(\frac{e_k(\mathbf{z}) - \phi e_{k-1}(\mathbf{z})}{\hat{\sigma}}\right)^2. \end{aligned} \tag{4.8}$$

If we used a different $\hat{\sigma}$ for each lag k, then this would also be represented in Equation 4.8. But as we do not, we have one single shared $\hat{\sigma}$ value. [5]

We estimate the values for ϕ and $\hat{\sigma}$ through their maximum likelihood estimates,

$$\phi = \frac{\frac{1}{K}\sum_{k=1}^{K} y_{k-1}y_k}{\frac{1}{K}\sum_{k=1}^{K} y_k^2}, \tag{4.9}$$

and

$$\hat{\sigma}^2 = \frac{1}{K}\sum_{k=1}^{K}\eta_k, \tag{4.10}$$

where η_k can be calculated based on Equation 4.7 after calculating ϕ. Since this is a combination of Gaussians and AR models, we name it **AR_log_gaussian**. [16]

All of our likelihood functions are summarized in Table 4.2 again. We also list in this table the log probabilities each likelihood function calculates when evaluating the results of the model run with the default parameters from Table 1.1. This table indicates that *scipy_log_gaussian* results in low probability densities, and the fewer batches are used, the higher the probability, so that *scipy_pre_mean_log_gaussian* leads to the highest probabilities. A lower probability generally results in a lower acceptance rate but also faster improvement for the error functions, as essentially only samples that decrease the errors are accepted.

4.4.2 Proposal Covariance Matrices

For our proposal distribution to make good proposals, the underlying proposal covariance matrix should represent the covariance between the model's parameters. Unfortunately, we only know the upper and lower boundaries of each parameter beforehand

Table 4.2: The names of our likelihood functions and their probabilities for the model output when run with default parameters.

Name	Probability with default parameters
scipy_log_gaussian	-17,054,185.32
scipy_post_mean_log_gaussian	-2,746.69
scipy_pre_mean_log_gaussian	-46.88
scipy_monthly_mean_log_gaussian	-258,184.35
scipy_yearly_mean_log_gaussian	-1,063.28
scipy_per_month_log_gaussian	-5,394.33
AR_log_gaussian	-31,027.25

and do not know their covariance. But we can try to fill the diagonal of the covariance matrix, i.e., the variance of the parameters, based on the range of their boundaries. Those variances can be scaled by some scalar s to allow for some customizability. For example, for *TT* with an upper bound of 4 and lower bound of -4, the variance is

$$\text{Var(TT)} = (4 - (-4))\, s. \tag{4.11}$$

We use three different proposal covariance matrices with s being **0.05**, **0.01** and **0.005**, respectively. For convenience, we name each matrix by its s value, e.g., 0.01. But those are likely suboptimal covariance matrices, for example, because they assume that the parameters are independent since the off-diagonal entries are zero. Thus, we additionally use the covariance matrices calculated from the samples obtained with other algorithm runs. These runs all use the DREAM algorithm with the same configurations except for using different likelihood functions shown in Table 6.1. We use the DREAM algorithm to create these additional covariance matrices, as the DREAM algorithm does not rely on a proposal covariance matrix.

We refer to the matrices based on abbreviated versions of the likelihood functions' names. For example,*yearly* and *scipy* refer to the matrices obtained when using the *scipy_yearly_mean_log_gaussian* and *scipy_log_gaussian* likelihood functions with the DREAM algorithm, respectively. We select four of these covariance matrices, namely the **scipy**, **yearly**, **AR**, and **pre_mean** covariance matrices, to save time and reduce the number of runs we perform. Those four matrices are the more extreme cases: The *pre_mean* covariance matrix has overall the largest entries, similar to how *scipy_pre_mean_log_gaussian* accepts samples more easily. The *yearly* covariance matrix has average entries, whereas both *AR* and *scipy* have small entries.

4.4.3 Evaluation

In addition to the combination of the seven likelihood functions and seven proposal covariance matrices (three scaled diagonal matrices and four matrices based on samples obtained from the DREAM algorithm), we also perform one run each with the likelihood

function and its corresponding covariance matrix obtained through DREAM, e.g., the *scipy_per_month_log_gaussian* function with the *per_month* covariance matrix. For this reason, some settings like the *scipy_pre_mean_log_gaussian* function with the *pre_mean* covariance matrix appear twice in our evaluation. Therefore, we end up with 56 different algorithm results in total for this section.

Execution time

The first noticeable fact is that all executions take almost the same time: the fastest with the *yearly* proposal matrix and the *scipy_yearly_mean_log_gaussian* likelihood function finished in 8843s ($\approx$ 2h 27 min), whereas the slowest uses the *pre_mean* proposal matrix and the *scipy_pre_mean_log_gaussian* function and finished in 9244s ($\approx$ 2h 34min).

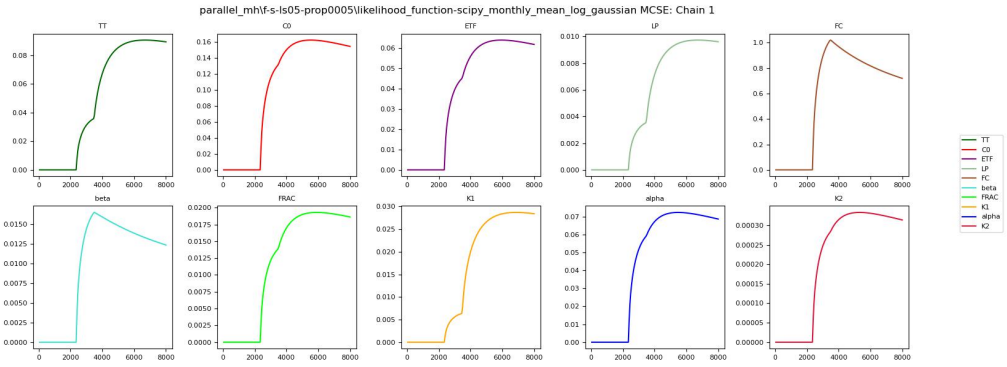
Interestingly, six out of the eight slowest all used the *AR* proposal matrix with their execution time ranging from 9176s to 9242s, while the other runs that use this proposal matrix were noticeably faster in comparison: the run using the *scipy_yearly_mean_log_gaussian* function finished in 8971s ($\approx$ 2h 29min), and when we reran the combination of *AR* proposal matrix and *AR_log_gaussian*, the execution finished in 8997s ($\approx$ 2h 30min). The latter result is especially noteworthy, as it uses the exact same settings as the eighth slowest. This time difference between the same setting suggests that the time differences in the execution time of the runs are not directly due to different parameters used for the algorithm. Instead, random outside factors like different proposed samples are the cause. But this result is to be expected, as the two parameters do not significantly affect the execution time: the different likelihood functions all work similarly, and the proposal covariance matrix only influences which parameters are proposed, but not how. Furthermore, the majority of the running time comes from the model run, which is not directly influenced by those factors.

MCSE

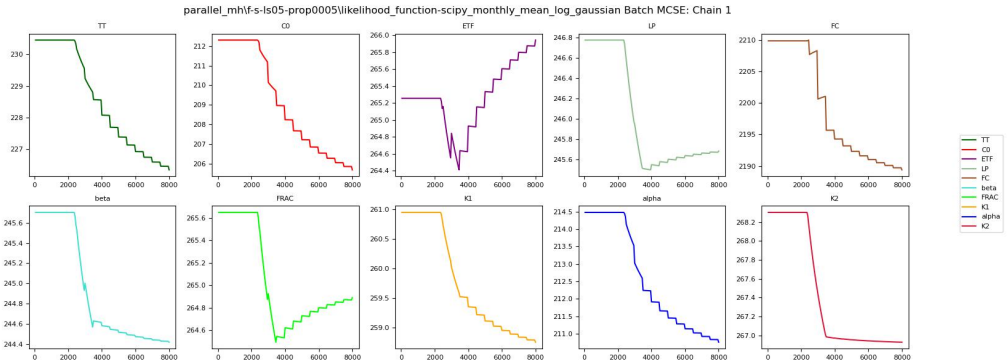
Figure 4.1 shows how the MCSE and batch MCSE results for chain 1 of the run using the *scipy_monthly_mean_log_gaussian* function and 0.005 proposal covariance matrix change as it obtains more samples. We select only one chain to save space, and because they all behave similarly for this run. Interestingly, the MCSE overall rises, whereas the batch MCSE decreases. However, the differences are minute, even over the entire run. Since this diagnostic provides less meaningful information compared to the error values, ESS, and R-convergence values, we focus less on the MCSE.

ESS

Another noticeable result is that the ESS is only calculable and still large enough to do an additional loop through the diagnostics with decreased sample size when using the *pre_mean* proposal matrix. Specifically, with the *scipy_log_gaussian* and the *scipy_pre_mean_log_gaussian* likelihood function, the max *esss_buithanh* is around 1000,



(a) MCSE



(b) Batch MCSE

Figure 4.1: MCSE results for chain 1 of the run using the 0.005 proposal covariance matrix and *scipy_monthly_mean_log_gaussian* function.

which allows for a still acceptable ESS. Notably, the mean and min *esss_buithanh* with the *scipy_pre_mean_log_gaussian* function are smaller by almost a factor of three than with the *scipy_log_gaussian* function. This difference suggests that the former function performs generally better but is held back by some dimensions where it performs comparably to the latter.

From the multi-chain ESS diagnostics, only bulk-ESS can calculate an estimate, and only when using the *pre_mean* matrix and the *scipy_pre_mean_log_gaussian* likelihood function. The bulk-ESS result is 31 for this setting, i.e., almost an ESS of 8 per chain. This result is very similar to its max *esss_buithanh* result of 1005, which also results in almost an ESS of 8 per chain. For any other setting, no ESS diagnostic for multiple chains could be calculated as no negative autocorrelation values were computed, signifying a too high correlation between samples. However, the setting with calculable bulk-ESS almost achieves negative autocorrelation values over multiple chains, only barely missing it for *FC* as we can see in Figure 4.2. The other settings also have their worst results for *FC*, but drastically worse as their autocorrelation for this dimension does not fall below 0.9. However, they also perform badly in other dimensions, as we see in Figure 4.3 as an example.

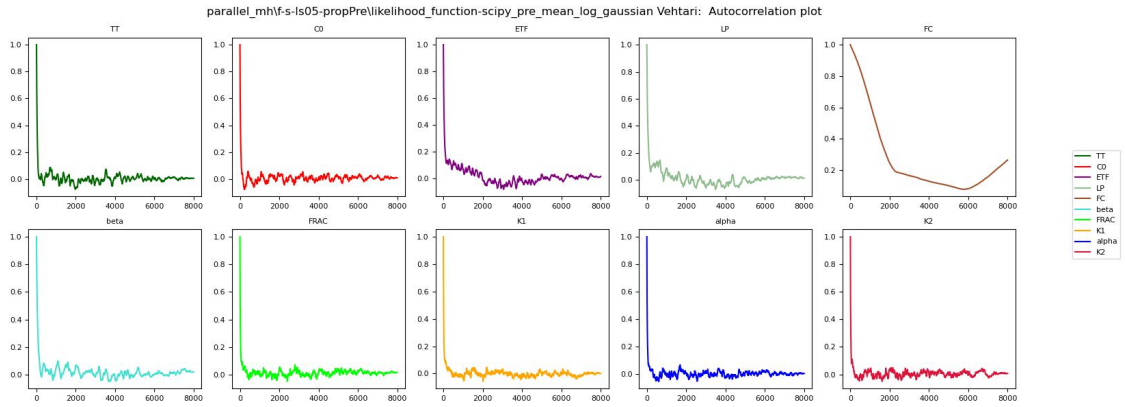


Figure 4.2: Best multiple chains autocorrelation in this section, obtained using the *pre_mean* matrix and the *scipy_pre_mean_log_gaussian* function

R-Convergence

Table 4.3 lists the best results of the R-convergence diagnostics. According to these results, we can only assume that one of the runs converges, namely the run using the *pre_mean* proposal matrix and *scipy_pre_mean_log_gaussian*, since it obtains a *rank/fold-*$\hat{R}$ value of 1.11 and is thus fairly close to 1.0. But the run with the second-best *rank/fold-*$\hat{R}$ value uses the same settings and still has a value of 1.6, which shows the inherent randomness of the MCMC algorithms. The performance of this setting is roughly reflected by the other R-convergence diagnostics as well, where it achieves the best and the third-best result in all other diagnostics.

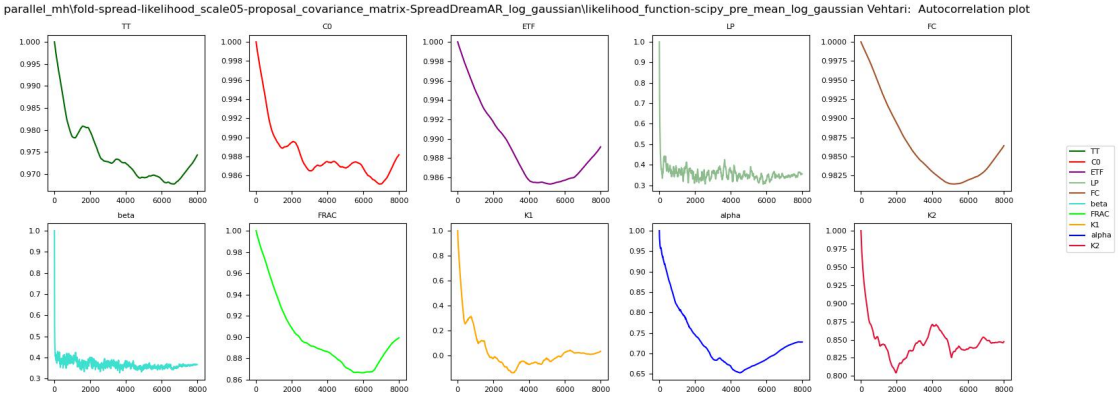


Figure 4.3: Average multiple chains autocorrelation in this section, obtained using the *AR* matrix and the *scipy_pre_mean_log_gaussian* function

Table 4.3: The R-Convergence results for the 6 best results according to *rank/fold-*$\hat{R}$.

Likelihood Function	Covariance Matrix	*rank/fold-*$\hat{R}$	mean *split-*$\hat{R}$	mean $\hat{R}$
scipy_pre_mean_log_gaussian	*pre_mean*	1.11	1.06	1.02
scipy_pre_mean_log_gaussian	*pre_mean*	1.60	3.95	2.41
scipy_log_gaussian	*scipy*	2.36	25.83	17.27
scipy_log_gaussian	*scipy*	2.38	29.11	14.92
scipy_pre_mean_log_gaussian	*yearly*	2.59	3.07	2.24

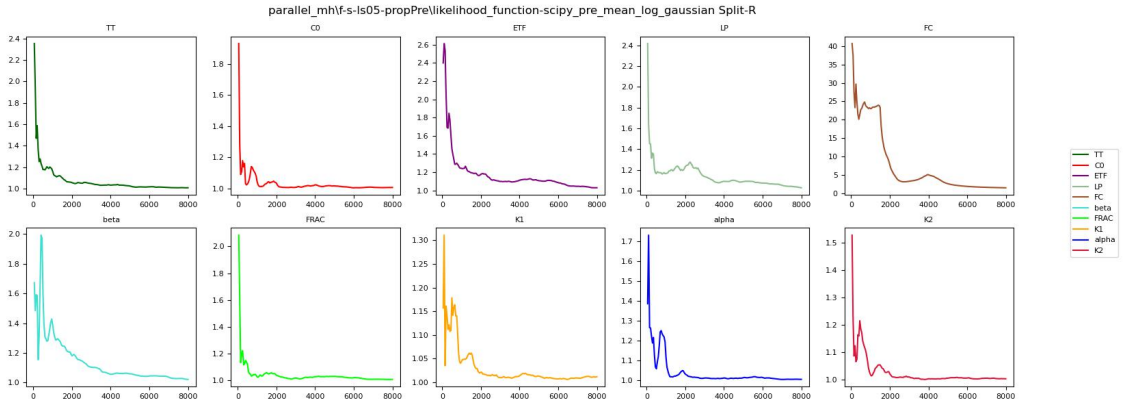


Figure 4.4: *split-$\hat{R}$*

The setting that has the second-best results for the other R-convergence diagnostics also uses the *scipy_pre_mean_log_gaussian* likelihood function, but with the *yearly* proposal covariance matrix, and has 2.59 as its *rank/fold-$\hat{R}$* value. The other settings have a *rank/fold-$\hat{R}$* value of 2.39 up to 8.98, with the 0.05 proposal matrix and *scipy_post_mean_log_gaussian* function being the worst setting for *rank/fold-$\hat{R}$*. The mean values of the other convergence diagnostics are at least 2.2 for these settings. It should be noted that the *FC* dimension vastly increases the mean of the other R-Convergence diagnostics, since it generally results in R-convergence values at least 10 times larger than the second largest $\hat{R}$ and *split-$\hat{R}$* value for another dimension. We can see this, for example, in Figure 4.4, which shows the *split-$\hat{R}$* value over the iterations for the run with the best *rank/fold-$\hat{R}$* value.

However, we see that even some of the best five in Table 4.3 have very large *split-$\hat{R}$* values, which makes their results for *rank/fold-$\hat{R}$* less reliable.

Diagrams

The ESS and R-convergence diagnostics both suggest that the *scipy_pre_mean_log_gaussian* likelihood function is the best likelihood function, as we only obtain adequate results by using this likelihood function. But if we examine the histograms in Figure 4.5 and Figure 4.6a of two runs using the *scipy_pre_mean_log_gaussian* likelihood function, we see almost uniform distributions for 3 dimensions, especially in the latter Figure. Unfortunately, this behavior fulfills the definition of uncorrelated samples required for good ESS values. The chains also all converge to the same area – the entire possible area within the boundaries in this case – as required for good R-convergence values. Therefore, those diagnostics' results are misleading. The time series plots in Figure 4.6b show this further as the behavior is the same regardless of the current iteration for three dimensions. Another way to discover this bad behavior is through the boxplot in Figure 4.6c. In this figure, we can see that the boxes as well as the medians are at the center of the parameter space, and the first and third quartiles are both about as far away from the median as from a boundary.

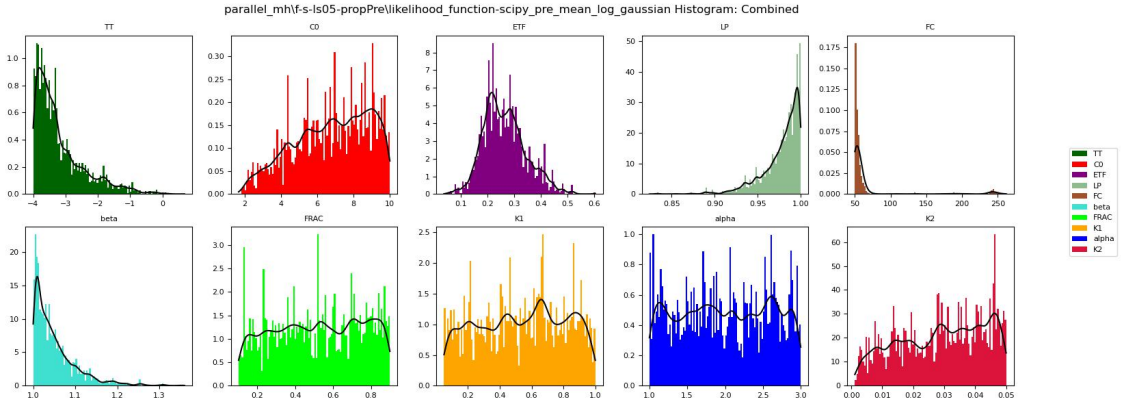
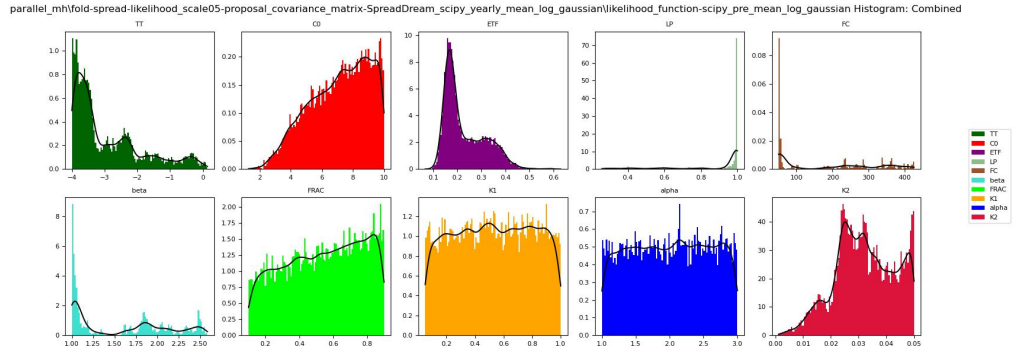
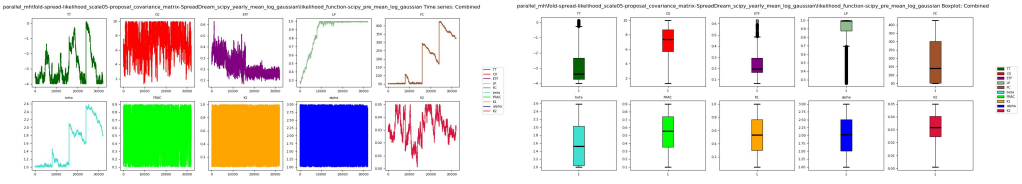


Figure 4.5: Histogram for the run using *scipy_pre_mean_log_gaussian* function and *pre_mean* covariance matrix with high acceptance rate



(a) Histogram



(b) Trace plot

(c) Boxplot

Figure 4.6: Example diagrams for a bad run due to very high acceptance rate for the run using *scipy_pre_mean_log_gaussian* function and *yearly* covariance matrix.

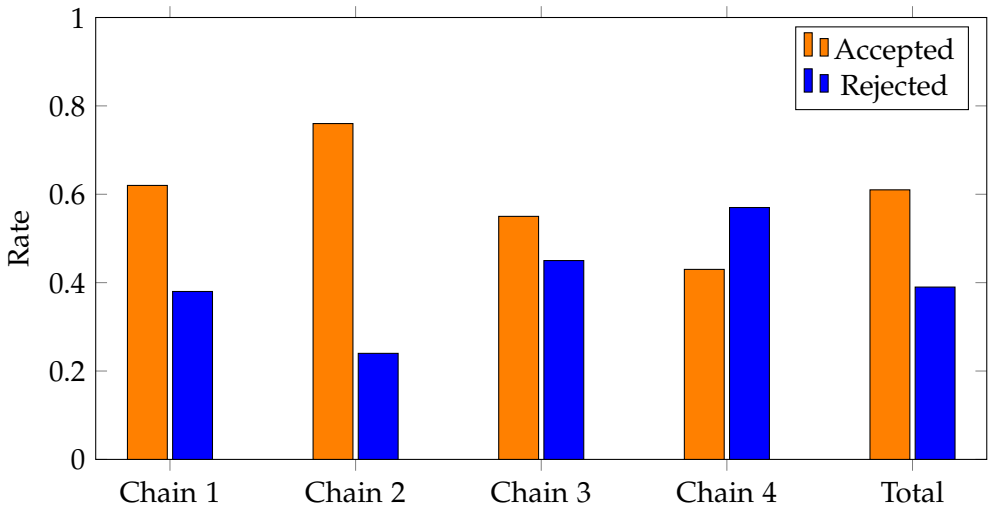


Figure 4.7: Acceptance rate of the run using *scipy_pre_mean_log_gaussian* and *yearly* covariance matrix

This bad performance is also represented by the error functions, where the mean MAE and mean RMSE of these runs are among the worst at around 42 and 78, respectively. It is especially noticeable in the case using the *scipy_pre_mean_log_gaussian* and *yearly* covariance matrix, which is also visible in the high acceptance rate of almost 60% visible in Figure 4.7. This high acceptance rate is presumably due to the likelihood function returning higher probabilities and the conservative proposals due to the relatively small proposal matrix.

Therefore, we first exclude such results where the histograms of multiple dimensions are either flat due to accepting too many samples, like in Figure 4.6a, or stay close to their starting location due to accepting too few samples, like in Figure 4.8.

Through this, we determine the *yearly* matrix to be the worst proposal matrix as all likelihood functions but the *scipy_yearly_mean_log_gaussian* function exhibit similar behavior to Figure 4.8 in multiple dimensions. For the *AR* and *scipy* proposal matrix, we also removed multiple results based on different likelihood functions, although not quite as many as with the *yearly* matrix. Surprisingly, we had to remove no results obtained through the covariance matrices that were based on a scaled difference between boundaries, although those do not diverge far from their starting value for the *FC* dimension. For this reason, we also do not test covariance matrices that were scaled even further down, as there is already barely any movement in one dimension.

Error Functions

Now, after the R-Convergence and ESS diagnostics are of no further help in deciding the best parameter, we make our decision based on the error functions. For this, we examine the five best options according to mean MAE, listed in Table 4.4. Those five

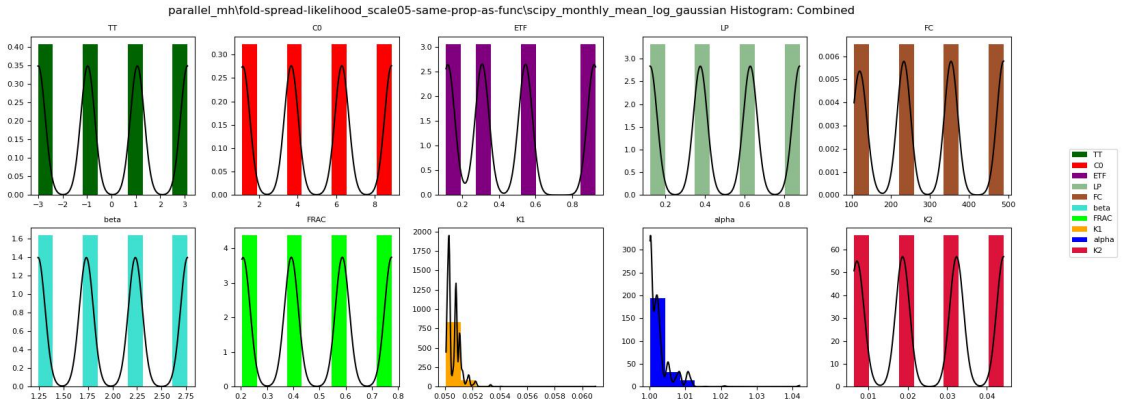


Figure 4.8: Histogram for the run using *monthly_log_gaussian* and the *monthly* covariance matrix with very low acceptance rate

best are also all among the eleven best for the other error functions as well.

Table 4.4: The Error function results for the 5 best results according to mean MAE.

Likelihood Function	Covariance Matrix	mean MAE	mode MAE	mean RMSE	mode RMSE
scipy_monthly_mean_log_gaussian	0.005	11.75	10.80	18.62	17.57
scipy_log_gaussian	0.01	11.76	10.91	18.71	17.75
scipy_log_gaussian	0.005	11.81	13.16	18.42	19.63
scipy_post_mean_log_gaussian	0.005	11.93	12.83	19.04	20.48
scipy_log_gaussian	0.05	12.10	11.49	18.78	18.40

As we can see, three of the five best results use the *scipy_log_gaussian* likelihood function, and three use the 0.005 proposal covariance matrix. The two best results are very close with roughly a difference of 0.1. The third-best is a bit farther behind but uses both the *scipy_log_gaussian* likelihood function and the 0.005 proposal covariance matrix and thus might be more consistent. This setting also has a slightly better mean RMSE, which further strengthens this assumption. As there is no clear best result, we will try those three settings in the next stage for further testing.

On a side note, the *AR_log_gaussian* function exhibits the behavior previously mentioned regarding a different test dataset for calculating the error values: The function results in generally better error values with the test set than with the training set, such as a test mean MAE of 10.24 and a mean MAE of 12.32 with the same model parameters. Hence, if we only decided based on the test errors, the *AR_log_gaussian* function would be the best, even though it is not among the best five according to the errors calculated on the training dataset.

It is noteworthy to emphasize that the error values vary considerably between the

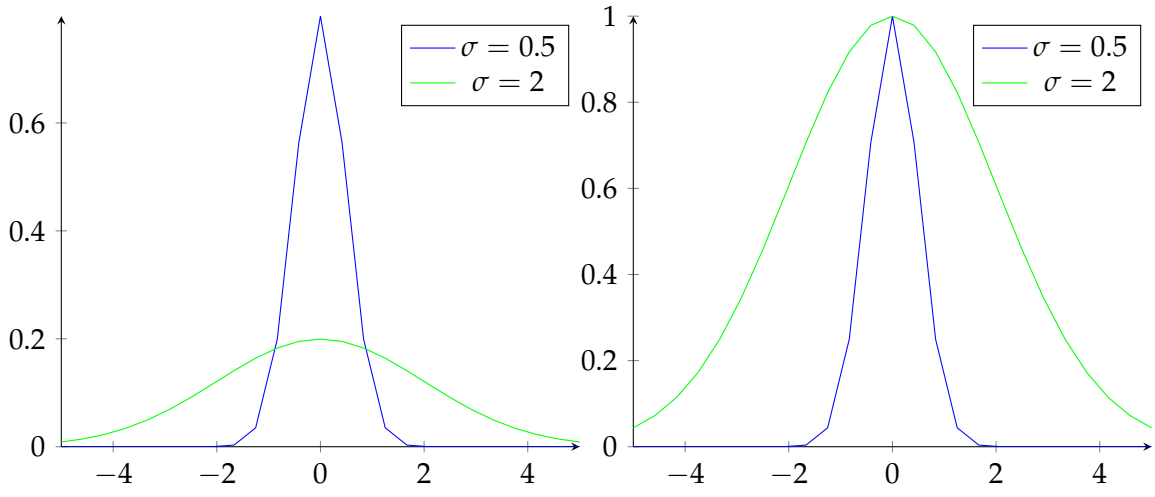
different runs, with, for example, the mean MAE values ranging from around 11 to 45. This variance makes it clear how important a fitting proposal covariance matrix and likelihood function are to obtain good posterior estimates.

4.5 Likelihood Scale

The likelihood scale is the σ from the definition of the Gaussian distribution given in Equation 4.5. Optimally, this option should be tested together with the previous two in the last section as it also directly influences the acceptance probability. But as this would dramatically increase the number of results to evaluate and the time required, we split them into two separate tests and examine them sequentially.

The likelihood scale affects how far off values can be from the mean and still result in relatively high probabilities. As can be seen in Figure 4.9a, a small σ translates to a small area around the mean that has high probability densities forming a peak, but values outside of this area essentially result in probability densities of 0. On the other hand, a larger σ means that a larger area around the mean has an overall smaller probability density, so that the peak is flatter. However, since we divide two likelihood probabilities to obtain the acceptance probability, the scale cancels itself out. Thus, we can treat the likelihood functions as normalized, and the likelihood scale affects only the width of the Gaussian likelihood as seen in Figure 4.9b. So, the larger σ is, the more samples we would expect to accept, and a smaller σ represents a stricter requirement.

We test six different values for σ: 5, 2, 1, 0.5, 0.1, 0.05. A value of 0.5 is what we had already used in the previous section.



(a) Scaled (b) Normalized

Figure 4.9: Example for different likelihood scale values for Normal distributions, left scaled, right normalized.

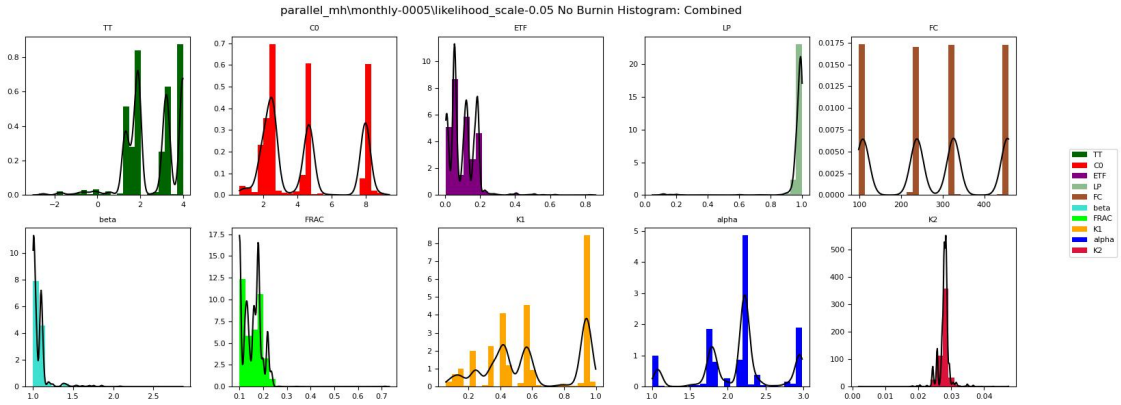


Figure 4.10: Histogram with relatively equally spread values in dimensions *C0*, *K1*, and *alpha*

Evaluation

The results for the Execution time, ESS, and R-convergence are similar to before, and show no clear correlation to the likelihood scale. In short, the execution times are similar for all settings, and the samples are too correlated so that the multi-chain ESS diagnostics can not be calculated. Furthermore, an additional run of the diagnostics with reduced sample size based on max *esss_buithanh* is meaningless since the average max *esss_buithanh* is 2112, or 1385 if we exclude those runs that accept no new sample in one chain and thus achieve the worst possible max *esss_buithanh*. The *rank/fold-*$\hat{R}$ values are also at least 3.40, with most lying between 4.0 and 6.0.

However, contrary to our expectations, we can not discover any correlation between the likelihood scale and the acceptance rate here. We assume the reason for this is that the likelihood functions are so strict that they outweigh the changes caused by the different likelihood scale values.

When looking at the histograms, we also see mostly similar behavior as before for most dimensions, especially for *FC*, where the samples stay close to their starting position. But, we additionally observe a similar behavior for some results in other dimensions, especially for dimensions *C0*, *K1*, and *alpha*, as can be seen in Figure 4.10. The box plot in Figure 4.11 further illustrates this, as the boxes for these dimensions are again quite large and the median is close to the center of the parameter space, representing that the first and third quartiles are far apart and the samples are equally spread. Other settings still obtain better results for these dimensions.

To be stricter in this evaluation than in the previous section, we already exclude those results that show bad convergence in one dimension besides *FC* from further evaluation. For this reason, we exclude the extreme likelihood scales of 0.05, 2, and 5. Furthermore, we remove results with a likelihood scale of 1 obtained with the *scipy_monthly_mean_log_gaussian* function and matrix 0.005. The error values support this decision: We again list the top 5 settings according to mean MAE in Table 4.5, none

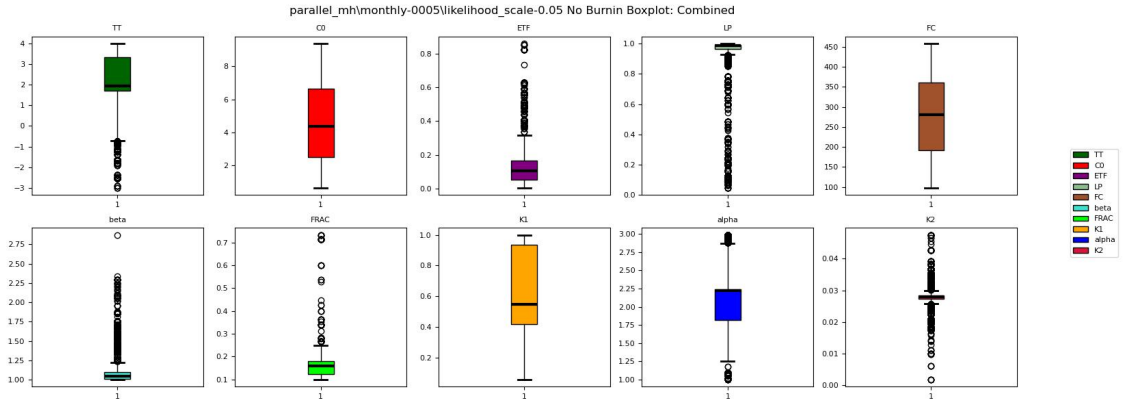


Figure 4.11: Box plot with relatively equally spread values in dimensions *C0* and *K1*

of which we have removed based on the diagrams.

Table 4.5: The Error function results for the 5 best results according to mean MAE.

Function	Cov Matrix	Scale	mean MAE	mode MAE	mean RMSE	mode RMSE
scipy	0.005	0.1	11.36	11.10	17.81	17.38
scipy	0.01	1	11.51	11.53	18.05	17.84
scipy	0.005	0.5	11.54	11.82	18.16	18.51
scipy	0.01	0.1	11.54	11.57	18.12	18.12
scipy	0.01	0.5	11.58	11.59	18.29	17.97

Since two of the best three use the *scipy_log_gaussian* function and the 0.005 matrix, we use these for further testing of the MH algorithm. Those parameters also lead to some of the best ESS and *rank/fold-*$\hat{R}$ values. Furthermore, this setting with a likelihood scale of 0.1 is the best according to all four error functions. It additionally achieves the best test mean MAE and test mean RMSE, which further proves its superiority over other parameter settings. Therefore, we use a likelihood scale of 0.1 in the further tests.

We do not discuss the multi-chain ESS values for the remainder of this chapter, as they are not calculable and thus irrelevant.

4.6 Starting Sample

4.6.1 Methods

Next, we test different methods to decide on starting locations for each chain. So far, we have used the *spread* method, where the starting samples are equidistantly spread across the whole bounded area for each chain. We used this so far as it introduces the least bias, but it has its downsides, as we see for the *FC* dimension. Another method we

test is to select samples uniformly within the boundaries, which we name *random*. This has the potential to return good initial samples, but can also return very bad samples. Furthermore, it is inconsistent and would have thus influenced the results if we had used *random* instead of *spread* in the tests in the previous sections.

Two more simple methods that assign the same starting sample to each chain, are the *default* and *prior mean* methods. As their names imply, they assign the default values for each dimension as in Table 1.1, or the mean of each dimension's boundaries, respectively.

The last method for the initial sample we use is the *close* method. The method first receives a base value for each dimension. It then draws the initial samples from a Gaussian distribution with that base value as the mean and a scale based on the dimension's boundary size. The resulting initial samples are then close to the base value. If the starting samples were to be out-of-bounds, we would move them inside the boundaries with the *eflect* method as it keeps values close to their nearest boundary. We describe *eflect* together with the other methods to handle out-of-bounds samples in the next section, Section 4.7. For the base value, we use four different values: the default values as in Table 1.1, and three values based on samples obtained in previous runs, namely the mean and two different modes. We use two different mode values, as those values differ when we examine the mode of all samples or when we determine the mode after removing the first 20% of samples due to burn-in. For example, for *FC*, the mode value with all samples is 122.5, whereas it is 51.4 after burn-in. For the mean values, we take the mean of samples after burn-in, as they reflect the data we evaluate in the end and are almost the same with or without burn-in. The mean and mode values we use for this are listed in Table 4.6.

Table 4.6: The mean and mode for each dimension based on the previous samples after burn-in.

Name	Mean Burn-in	Mode Burn-in	Mode All Samples
TT	1.27	-2.85	1.54
C0	4.77	0.41	2.42
ETF	0.24	0.04	0.26
LP	0.80	0.08	0.90
FC	246.51	51.40	122.47
beta	1.45	1.00	1.09
FRAC	0.28	0.10	0.10
K1	0.33	0.05	0.81
alpha	1.57	1.01	2.34
K2	0.02	0.005	0.028

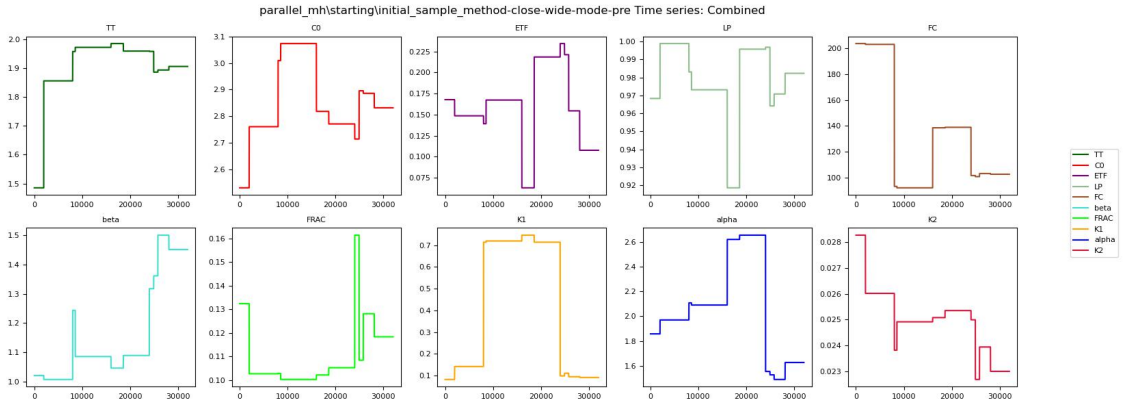


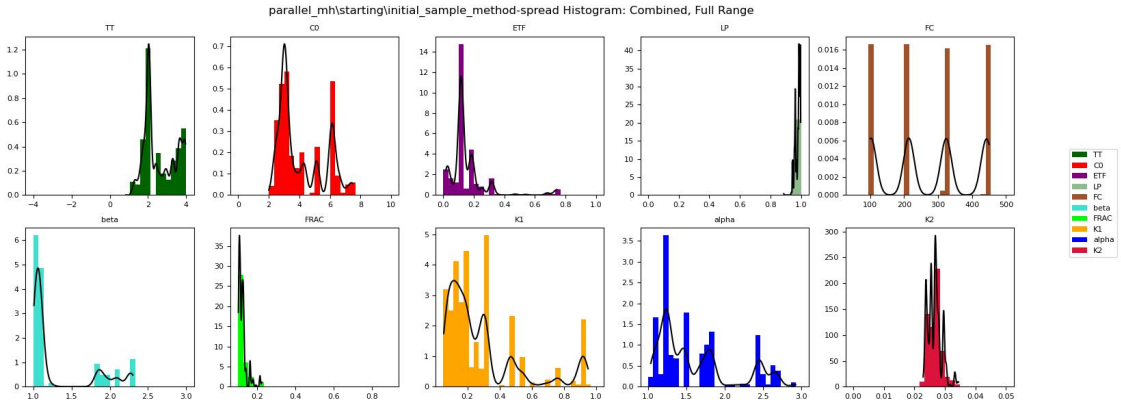
Figure 4.12: Trace plots of the run using the *close mode before burn-in* starting sample method

4.6.2 Evaluation

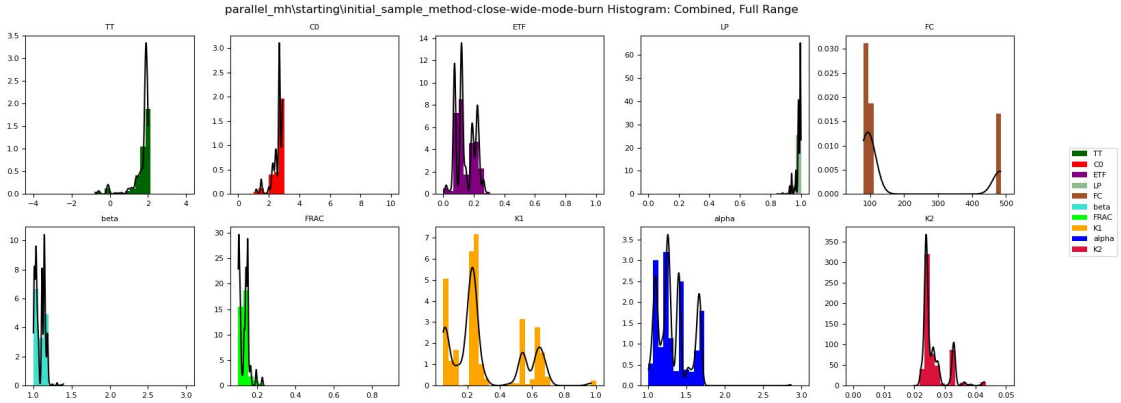
The results for the execution time, ESS, and R-convergence are again similar to before: similar running times with a difference of at most three minutes, max *esss_buithanh* values of at least 1162 and an average of 1324, and *rank/fold-*$\hat{R}$ values of at least 3.12. Using starting values close to the mode before burn-in offers the smallest max ESSS value, 1162, but it also returns the largest *rank/fold-*$\hat{R}$ value, 6.08. This starting value also has the smallest mean MCSE, but that is presumably because only 10 samples were accepted after burn-in among all four chains, as can be seen in Figure 4.12, and thus does not indicate good convergence.

But this time, the error functions' results are inconclusive as well as they differ by at most 1.6 each. Furthermore, the error functions over the result from using the *default* values as starting values have the smallest errors, but using the *prior mean* offers the worst error values for all but the mode RMSE, where this starting method is the third-worst. This is interesting, as *default* and *prior mean* use almost the same starting values, with only a difference of 175 in *FC* and a slight difference of 0.025 in *K1*.

Since there is no clear best choice, we continue using the *spread* starting value. Most importantly, we keep this initial sample method because a fundamental advantage of using multiple chains is to explore different areas within the dimensions' boundaries by starting each chain at different areas within the boundaries. This independence of chains is also needed to make the R-convergence results more reliable. This is not given at all by *default* and *prior mean*, and only to some degree with *close*, whereas *spread* uses unbiased starting values. Additionally, it has average results in most diagnostics, and not the best in one diagnostic and the worst in another, so it is more consistent. Furthermore, as can be seen in Figure 4.13, the results from *spread* converge to a similar distribution as starting close to the mode after burn-in results in, especially in *LP*, *beta*, *FRAC*, and *K2*. Hence, the starting samples are not too influential since the chains already converge during burn-in.



(a) *spread*



(b) *close mode after burn-in*

Figure 4.13: Histograms after burn-in with starting values equidistantly spread and close to the mode after burn-in.

4.7 Boundary Handling

As mentioned previously, we must ensure that the samples are within the boundaries for every dimension to avoid errors while running the model. We first introduce the four different methods we use to guarantee that the model is only run with samples within the boundaries before discussing the results of the tests.

4.7.1 Methods

Ignore

The first method is to directly reject out-of-bounds samples, which we thus name *ignore*. This can also be seen as calculating the acceptance rate as normal, but receiving a likelihood and thus acceptance rate of 0 due to the uniform prior. It is the simplest method, and can offer a speed up as we do not have to run the model for the current proposal if we already know it will be rejected beforehand. [7]

Bound

The *bound* method moves samples to the boundary of each dimension that they crossed. For example, a sample is within the boundaries for all dimensions but *TT*, where it has a value of -5 and is thus below the lower bound of -4. Then the sample would keep the same values in all dimensions but *TT*, for which it would be set to -4. While this method is also simple, it has the potential downside of accepting far more samples at the boundary than would be correct, especially with a symmetric proposal distribution as we use: If the current sample is at a boundary, then half of the proposals would be out of bounds and thus moved to the boundary value again. Thus, the boundary value would be accepted and added to the accepted samples again. For the other half of the proposals, there is only a chance for them to be accepted. If they are rejected, then we add the boundary value to the accepted samples again. [5]

Eflect

Another method is to mirror the out-of-bounds samples at the broken boundary. Using the previous example, the value for *TT* is -5 and thus 1 below the lower boundary, so after mirroring it, it is 1 above the lower boundary, at -3. If this mirroring would cause the sample to break the other boundary, then we repeatedly reflect at the broken boundary until no boundary is broken anymore. But this would require the sample to exceed a boundary by at least the size of the entire boundary, which is unlikely to happen with a proper proposal distribution. [5]

Fold

Both the *Bound* and *Eflect* methods violate detailed balance, which is necessary for our MCMC algorithm to converge to the target distribution. But we still keep them for testing as they are commonly used. Additionally, detailed balance is only needed in theory to guarantee convergence, but not strictly necessary to obtain good results similar to the target distribution. [5]

The *fold* method does preserve detailed balance, though, which is why we have used it so far in the previous tests. In this method, we treat the lower and upper bounds as connected, so that if a sample breaks one bound, it is mapped towards the other bound. Using the previous example again, we would map the value for *TT* to 1 below the upper bound of 4, i.e., to 3. The same results would happen if the initial value for *TT* is -13, -21, ..., that is $(-5) + 8x$ since *TT*'s boundary has a size of 8. Thus, this method moves samples to the other side of the parameter space, which can be problematic if the target distribution is near one boundary but far from the other boundary. Then, many proposals would be out-of-bounds, moved to the other boundary, and thus likely rejected since the other boundary has low probability densities. [5]

4.7.2 Evaluation

Execution time

The most noticeable result for the different methods to ensure the samples stay within the boundaries is that the *ignore* method is significantly faster, finishing in 2737s (46min), whereas the other methods all require around 9000s (2h30min) as before. But that is already where the significant results end.

Error functions and MCSE

The *bound* method achieves the best mean MAE of 10.95 and mean RMSE of 17.31. The other three methods achieve a mean MAE of around 12.50 and a mean RMSE of around 19.50, with a difference below 0.5 among their results, and with *fold* achieving the lowest errors. It is reversed for the mode values where *fold* and *bound* have the worst results, especially for mode MAE, where they achieve values around 11.63 and the other two methods around 10.60. For the mode RMSE, *fold* is still the worst with 18.49 and thus a difference of 0.99 to *bound* as the next worst. However, *eflect*'s mode RMSE of 17.31 is very close to *bound*'s result, and *ignore* is significantly better with a mode RMSE of 16.89.

Therefore, the error functions give no conclusive results as *ignore* has the best results using the mode of samples and *bound* when using the mean to rerun the model. The latter results suggest again that the modes of the target distribution in some dimensions are close to a boundary.

Their MCSE values are also very close, ranging from 0.13 to 0.17.

ESS

The samples obtained from using *eflect* and *bound* are very correlated and thus have a max *esss_buithanh* of 4001 and 4000, respectively. This indicates that all samples of at least one chain have the same value in at least one dimension. The *ignore* and *fold* methods achieve good ESS results with a max *esss_buithanh* around 1150.

R-convergence

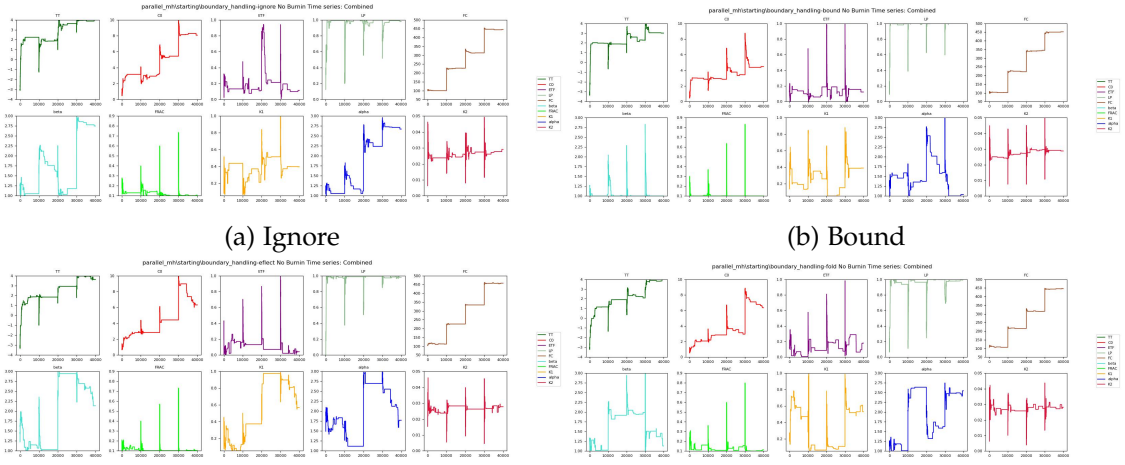
For the R-convergence values, *eflect* has the worst *rank/fold-*$\hat{R}$ values at 5.23, and *ignore* and *fold* have similar values with 4.51 and 4.44, respectively. The best result has *bound* with 3.67. However, the *bound* method causes the chains to converge to a boundary, even if it is not the actual target distribution. This can also be seen in the trace plots in Figure 4.14, where most chains converge to the same area. However, *bound* converges more quickly for *LP*, *beta*, and *FRAC*, where the target distribution seems to be close to a boundary. But as the target distributions are likely only close to a boundary, but not exactly at that boundary for some dimensions, the *bound* method overshoots. So, the relatively small $\hat{R}$ value might only indicate convergence to the boundary, but not to the actual target distribution, and is thus unreliable.

Because some dimensions converge close to a boundary, the *ignore* method is much faster than the other methods, as many proposals are out-of-bounds and thus rejected before running the model, saving time. The probability density is seemingly very high near one boundary and very low near the other boundary for some dimensions. Thus, when the current sample is in the high probability density area, the *fold* method maps out-of-bounds proposals to the other boundary, i.e., the low probability density area, which will likely cause them to get rejected as well. So, in this case, the *fold* method accepts and rejects similar proposals to the *ignore* method, with the downside of always running the model first, causing a significant time difference. However, it is not guaranteed that we reject out-of-bounds proposals that were transformed due to the *fold* method, whereas *ignore* is guaranteed to reject such proposals, which is why *fold* has more unique samples.

There were no significant and consistent differences in the error functions, but the *fold* and *ignore* methods behave better for the R-convergence and ESS diagnostics. Thus, we use both the *fold* and *ignore* methods while testing in the next section, where we try out different numbers of chains.

4.8 Number of Chains

We use 2, 4, 6, and 8 chains with both the *fold* and *ignore* methods. While the *ignore* method seems better than the *fold* method due to its faster execution time with similar results, we expect the execution time to be the most significant change when using a diverse number of chains. Since the *fold* method is more consistent in its execution times,



(a) Ignore

(b) Bound

(c) Eflect

(d) Fold

Figure 4.14: Trace plots before burn-in with all boundary handling methods.

it allows us to observe changes more clearly. Furthermore, we increase the number of iterations per chain to 20,000 when using the *ignore* method to see any significant improvements, as that should still be faster than most of the previous test runs we executed before. Additionally, the starting positions change with the number of chains because of the *spread* method.

In Table 4.7, which compares the execution times of the different runs, we can make three observations:

1. The *ignore* method is still faster even when doing twice as many iterations as the *fold* method.

2. The samples per second, i.e., the number of completed iterations per second, increases with the number of chains, as we expect.

3. The rate at which the completed samples per second increase is roughly similar for both *ignore* and *fold*, with *ignore* increasing slightly quicker.

What this table does not show though is that the execution times for each chain varies a lot more for the *ignore* method: while the execution time between chains with the *ignore* method differ by up to 30 minutes, they differ by at most 2 minutes with the *fold* method, which shows *fold*'s consistency we mentioned before.

ESS

The max *esss_buithanh* values are roughly twice as large when doing 20,000 iterations with *ignore* than with *fold* and 10,000 iterations, which leads to similar ESS. This difference is probably because the former obtained a similar number of unique samples in a larger total number of samples, affected by the comparatively longer burn-in phase of 4,000

Table 4.7: Execution times relative to sample sizes.

	Number of Chains	Full Execution Time	Samples per Second	Relative to 2 Chains
Fold	2	6881.95	2.32	1
	4	8991.17	3.56	1.53
	6	10903.16	4.40	1.89
	8	13097.77	4.89	2.10
Ignore	2	2955	10.03	1
	4	3850.04	16.62	1.54
	6	4448.34	21.58	1.99
	8	5325.336	24.04	2.22

samples. Additionally, due to the larger number of iterations, a lag increase of one has less influence on the sum in Equation 3.12.

Using more chains also means that it is more likely to obtain chains stuck at one value and accept no new samples, which causes the worst max *esss_buithanh*. This is the case with 8 chains for both options and 6 chains with *ignore* and 20,000 iterations. Besides this fact, the number of chains does not affect the max *esss_buithanh* as they achieve mostly similar values between 1111 and 1491 for 10,000 iterations and 2812 and 2862 for 20,000 iterations, as we expect, since the chains are independent.

R-convergence

The *rank/fold-*$\hat{R}$ values increase as the number of chains increases, from 3.09 to 8.17 for *fold* and from 3.15 to 6.08 for *ignore*. This makes sense as a higher number of chains means that more chains must converge to the same behavior, making the diagnostic stricter and thus more difficult to achieve low values for.

Error functions

For the error functions results, 6 chains achieve the best results for both boundary handling methods according to mean MAE with 11.59 for *fold* and 11.85 for *ignore*. With *ignore*, using 6 chains results in the smallest values for the other three error functions as well. With *fold*, 6 chains additionally have the best result only in mode RMSE, as 2 chains result in better mode MAE and 4 chains in a slightly better mean RMSE. But overall, they differ by at most 1.15, a negligible amount considering the inherent randomness in MCMC results.

6 chains offer slightly smaller error values. The increased complexity of multiple chains makes the R-convergence values more reliable, and the increased values are acceptable in return. The increased number of iterations does not seem to have any

significant influence besides longer execution time and larger ESSS. So, we use the *ignore* method with 10,000 samples per chain as before, but use 6 chains instead.

4.9 Burn-in Length

The burn-in length does not affect the algorithm run directly, but how the results are processed. So, the execution time and acceptance rate of the algorithm are irrelevant for this setting, as they stay the same. We reuse the result from the previous section with *ignore* and 6 chains to test different burn-in lengths: no burn-in, 20%, 50%, and 80% of the total size.

Unique Samples

The natural result we observe is that the number of unique samples decreases with longer burn-in lengths, but it is interesting how quickly they decrease. We have 243 unique samples when doing no burn-in, but only 32 when discarding the first 20% of samples, and only 10 when we discard 80% of samples. This signifies that more samples are accepted in the beginning, which also proves the purpose of burn-in: more proposals are accepted in the beginning, because comparatively many proposals are better than currently accepted samples, and the number of unique samples increases. This is no longer the case during the last 80% where the samples have already converged in multiple dimensions and thus fewer proposals are accepted. This can also be seen in the mean MCSE values, where the value without burn-in is more than twice as large than with 20% burn-in length, and almost six times larger than with 80% burn-in length.

Error functions

The results for the error functions are also very similar. Specifically, when using the mode of samples for a model run, no burn-in and 20%, as well as 50% and 80%, have the same MAE and RMSE values of 10.80 and 10.87, and 17.21 and 17.83, respectively. This shows that the value that appeared the most overall is among the first half of the samples, and therefore changes if we remove the first half. Furthermore, the mean RMSE values improve as we discard more samples from the beginning. The error values show a similar trend for the mean MAE values, although a burn-in length of 50% results in a value of 11.756 and is thus slightly smaller than using a burn-in length of 80%, which results in a mean MAE value of 11.761. But, as the difference in those values is insignificant, we can assume that the mean of the samples moves closer to the target distribution.

ESS

For the max *esss_buithanh* results, using no burn-in technically only has the second smallest value with 3342, but the other three burn-in lengths all result in the worst

possible ESSS of 8000, 5000, and 2000, respectively. That is caused by one chain accepting no new samples after the first 20% of iterations, so all samples are the same value for that chain. This results in the worst ESSS based on our chosen maximum required autocorrelation value of 0.5 to be treated as uncorrelated. Without burn-in, there are still comparatively many unique samples and thus more independent samples, so the autocorrelation values reach the required value more quickly. But this still only results in a small ESS of 5 per chain, which is too small to be of use.

R-Convergence

The *rank/fold-*$\hat{R}$ values increase from 4.41 to 16.76 as we discard more samples. That is because $\widehat{var}$ stays nearly constant, but the variance within the chains decreases as they get stuck on a few values (as seen by the small number of unique samples and MCSE). Therefore, the $\hat{R}$ values all increase according to Equation 3.8.

Since the R-convergence values increase with longer burn-in, but the error functions decrease, we want neither of the extremes of no burn-in and 80% burn-in. We determined no significant difference between 20% and 50% burn-in besides the *rank/fold-*$\hat{R}$ value, where the former is notably better, so we use 20% burn-in length.

4.10 Parallel MH against Basic MH

We also perform one execution of the Basic MH algorithm, i.e., with only one chain.

As expected, the execution time further decreases than with two chains by about 15 minutes, and the acceptance rate stays similar. We obtain a similar mean *esss_buithanh* as with multiple chains, which is for one chain also the max and min *esss_buithanh*. The mode error values are slightly worse by 0.3 for MAE and 0.2 for RMSE than the results we obtained when using 6 chains, but the mean error values improve by 0.8 for MAE and 1.1 for RMSE.

However, using one chain is also less consistent, as one coincidentally good chain can not balance out a bad chain. Furthermore, as this disables the use of the other ESS diagnostics and all R-convergence diagnostics, we continue to use the parallel version with 6 chains.

4.11 Summary

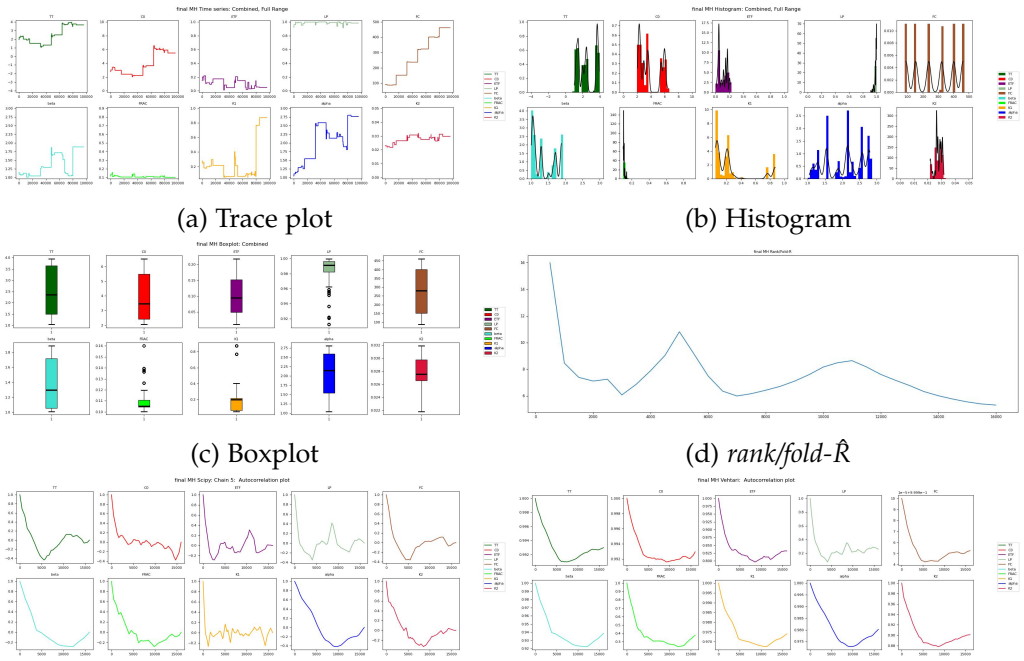
Table 4.8 lists the final parameters that we have decided on through the previous tests. The results we obtained with those parameters are listed in Table 4.9 and Figure 4.15. For the single chain autocorrelation plot, we selected only chain 5 as it accepted the most unique samples, and to save space. Chain 2 accepted no new samples, so its autocorrelation plot is just a straight line, and the other chains are in between those two cases.

Table 4.8: The optimal parameters of the MH algorithm according to our tests.

Parameter Name	Value
Likelihood Function	*scipy_log_gaussian*
Proposal Covariance Matrix	0.005
Likelihood Scale	0.1
Boundary Handling	Ignore
Starting Sample	Spread
Number of Chains	6
Burn-in Length	20%

Table 4.9: The final results using the mean and mode of samples obtained through the settings in Table 4.8.

Name	Mean	Mode
TT	2.46	1.50
C0	3.83	2.41
ETF	0.10	0.17
LP	0.99	1.00
FC	277.11	152.17
beta	1.37	1.04
FRAC	0.11	0.10
K1	0.26	0.22
alpha	2.06	1.54
K2	0.028	0.027
MAE	11.85	10.80
RMSE	18.53	17.21
mean *esss_buithanh*	3161	
rank/fold-$\hat{R}$	5.31	
mean *split-*$\hat{R}$	21.61	



(a) Trace plot

(b) Histogram

(c) Boxplot

(d) *rank/fold-*$\hat{R}$

(e) Single-chain autocorrelation

(f) Multi-chains autocorrelation

Figure 4.15: Diagrams for the final MH run.

We see in this chapter that the proposal covariance matrix and likelihood function have a major influence on the results. However, most other parameters have barely any influence besides the *ignore* method for boundary handling, which decreases the execution time. The acceptance rate is also very low, with rates below 10% for all settings besides the *yearly* proposal matrix and *scipy_pre_mean_log_gaussian*, and the *scipy* proposal matrix with *scipy_pre_mean_log_gaussian*, for which the error functions return bad results. Especially the former has a too high acceptance rate as we already discussed in Figure 4.6a. Thus, the average acceptance rate is below 3%.

However, at the beginning of the executions, comparatively many samples are accepted, which suggests that the proposal covariance matrix is suitable for the beginning. But as the acceptance rate decreases, the covariance matrix needs adjustments after some iterations. We address this issue in the next chapter, Chapter 5.

Furthermore, the multi-chain diagnostics are either not calculable (ESS) or show bad results (R-convergence). But this is not unexpected, as we use a straightforward parallelization method for the simple and basic MH algorithm to obtain multiple chains.

We also notice a trend with the accepted samples for a few dimensions, such as *beta* values converging close to the lower boundary, or *LP* values to the upper boundary in almost all runs, which suggests that we have found the target distribution for those dimensions.

5 Delayed Rejection Adaptive Metropolis algorithm

5.1 Introduction to the Delayed Rejection Adaptive Metropolis algorithm

The next algorithm we examine is the Delayed Rejection Adaptive Metropolis (DRAM) algorithm, which originally also works with a single chain. It is a refinement of the MH algorithm, combining the originally independent schemes of Delayed Rejection (DR) and Adaptive Metropolis (AM). We parallelize it in the same straightforward way as the MH algorithm. As DR and AM change different parts of the MH algorithm, they can easily be combined without any additional changes. Due to the changes the AM scheme makes, the DRAM algorithm only works with a Gaussian proposal distribution. Furthermore, since none of the ESS diagnostics for multiple chains are calculable, we do not discuss them further in this chapter. [4]

5.1.1 Delayed Rejection

The idea behind the DR approach is simple: instead of immediately continuing with the next iteration once we reject a proposal, we add another stage. In this second stage, we create another proposal that is based on both the last accepted sample and the last rejected proposal in this iteration with a scaled proposal covariance matrix. Generally, the proposal covariance matrix is scaled down in later stages to decrease the variance, as proposals close to already accepted samples are more likely to be accepted. Furthermore, this allows the original matrix to be larger to enable bolder proposals first to explore the state space faster through the first stage. How much the matrix should be scaled down is undecided; some decrease it by only 0.5, whereas others do so by 0.01. We initially use a value of 0.1, though we test additional values in Section 5.7. Theoretically, this can continue with a third proposal based on the last two proposals and the last accepted sample if we reject the second proposal, and so on until we accept a proposal. But, this is more time intensive, as we have to run the model for each proposal, and causes too small changes if we scale the proposal covariance matrix down at each stage. Therefore, we only evaluate at most a second proposal, as is common for the DRAM algorithm. [4]

The proposal distribution and acceptance probability for the first proposal are still the same as in the MH algorithm in Equation 4.3. However, to differentiate between a proposal distribution using the original covariance matrix and one using a scaled covariance matrix, we add a subscript if needed: q_1 uses the original covariance matrix

and q_2 the scaled version. The definition of the second stage proposal distribution is unclear in the paper explaining the DRAM algorithm: On one hand, the proposal distribution should be dependent on the last accepted sample and the last rejected sample, but on the other hand, the paper also states that the proposal distribution is symmetric. [4]

To fulfill the former requirement, we can take the mean of the last accepted sample and last rejected proposal as the mean of the Gaussian proposal distribution to sample the second proposal. Then, $q(z_k \mid z', z'')$ signifies that we examine the probability density of z_k for a Gaussian distribution, which uses the mean over z' and z'' as its mean. But with this approach, the proposal distribution is no longer symmetric since $q(z_k \mid z', z'') \neq q(z'' \mid z', z_k)$ ($q(z'' \mid z', z_k)$ is generally larger).

If we want the proposal distribution to stay symmetric, we could define the second stage proposal distribution as $q(z'' \mid z_k, z') = q(z'' \mid z_k)$. Thus, we sample the second proposal from a Gaussian proposal distribution with the last accepted sample as its mean and ignore the last rejected proposal.

However, we could not think of a way to use a Gaussian proposal distribution – which has only a mean and covariance matrix as parameters – to be symmetric and dependent on multiple samples. Therefore, we test both approaches in Section 5.4.

The acceptance probability for the second sample z'' is defined as

$$\begin{aligned} \alpha_2(z'' \mid z_k, z') &= \min\left\{1, \frac{p(z'')q_1(z' \mid z'')q_2(z_k \mid z', z'')[1-\alpha_1(z' \mid z'')]}{p(z_k)q(z' \mid z_k)q_2(z'' \mid z', z_k)[1-\alpha_1(z' \mid z_k)]}\right\} \\ &\left(= \min\left\{1, \frac{p(z'')q_1(z' \mid z'')[1-\alpha_1(z' \mid z'']}{p(z_k)q(z' \mid z_k)[1-\alpha_1(z' \mid z_k]}\right\}\right), \end{aligned} \tag{5.1}$$

where the definition in parentheses holds if the proposal distribution is symmetric. If we put the posterior from Equation 2.1 as the target distribution p, then the equation above transforms into

$$\begin{aligned} \alpha_2(z'' \mid z_k, z') &= \min\left\{1, \frac{\pi(y|z'')q_1(z' \mid z'')q_2(z_k \mid z', z'')[1-\alpha_1(z' \mid z'']}{\pi(y|z_k)q(z' \mid z_k)q_2(z'' \mid z', z_k)[1-\alpha_1(z' \mid z_k]}\right\} \\ &\left(= \min\left\{1, \frac{\pi(y|z'')q_1(z' \mid z'')[1-\alpha_1(z' \mid z'']}{\pi(y|z_k)q(z' \mid z_k)[1-\alpha_1(z' \mid z_k]}\right\}\right). \end{aligned} \tag{5.2}$$

As before, to avoid numerical instability, we work with the log version of Equation 5.2. Because $q_2(z_k \mid z', z'')$ is generally smaller than $q_2(z'' \mid z', z_k)$, the non-symmetric version results in a lower acceptance rate in general, but depends on multiple proposals, which is a fundamental part of DR. In addition to the higher acceptance rate through the symmetric version of the equation, the symmetric proposal distribution also proposes new samples closer to an already accepted sample, which generally further increases the acceptance rate. [4]

5.1.2 Adaptive Metropolis

As we showed in the previous chapter, the proposal covariance matrix is very important for the MH algorithm to achieve good results. In general, a large proposal covariance matrix in the beginning is good as it allows us to perform larger steps between proposals. Thus, we can explore more of the state space quickly to find peaks, i.e., areas with high probability densities. But after finding such an area, which is likely small, it is useful to take smaller steps to stay within that area. For this to reliably happen, the proposal covariance matrix has to change to represent the lower variance. However, to make sensible adjustments and not random changes, we should include the previously accepted samples. This would make the algorithm no longer Markovian and not reversible, which means that we are no longer guaranteed to converge to the target distribution. [4]

Nonetheless, the AM algorithm manages to preserve the target distribution while adapting the proposal covariance matrix. After a non-adaptation period has passed, during which the covariance matrix stays at its initial value, the AM algorithm calculates a scaled version of the covariance matrix C over previous samples

$$C_k = s_d \text{Cov}(z_1, \ldots, z_{k-1}) + s_d \epsilon I_d, \tag{5.3}$$

where d represents the number of dimensions of the problem (10 in our case), s_d is a scaling parameter defined as s_0^2/d where s_0 can be freely chosen but is typically set to 2.4, ϵ is a very small value to ensure singularity of the covariance matrix, and I_d is the d-dimensional identity matrix. The length of the non-adaptation period is given by k_0, and can also be freely chosen. $\text{Cov}(z_1, \ldots, z_{k-1})$ is defined as

$$\text{Cov}(z_1, \ldots, z_{k-1}) = \frac{1}{k}\left(\sum_{i=1}^{k}(z_i z_i^T) - (k+1)\bar{z}_k \bar{z}_k^T\right), \tag{5.4}$$

where $\bar{z}_k = \frac{1}{k}\sum_{i=1}^{k} z_i$, i.e., the mean over the first k samples, and z_i are column vectors (so that $z_i z_i^T$ is a matrix). The mean $\bar{z}_k$ and the covariance matrix C_{k+1} have the following recursive formula:

$$\begin{aligned} \bar{z}_k &= \frac{\bar{z}_{k-1}(k-1) + z_k}{k} \\ C_{k+1} &= \frac{k-1}{k} C_k + \frac{s_d}{k}\left(k\bar{z}_{k-1}\bar{z}_{k-1}^T - (k+1)\bar{z}_k \bar{z}_k^T + z_k z_k^T + \epsilon I_d\right). \end{aligned} \tag{5.5}$$

This way, the covariance matrix can be updated with each new sample without loading every previous sample. However, it is suggested that k_0 not only defines the length of the non-adaptation period in the beginning, but also the length between each update, as doing updates with every new sample adapts the covariance matrix too strongly. So, we store the accepted samples between updates and then do multiple update steps according to Equation 5.5 for each new sample at once. [4]

5.2 Initial Settings

The starting values we use for the parameters besides the likelihood function and proposal covariance matrix are listed in Table 5.1. We use mostly the same initial settings as for the MH algorithm, except for the boundary handling method, where we use the *ignore* method due to the significant execution time speed up with no significant other changes we have observed in the last chapter with the MH algorithm.

Table 5.1: The other starting parameters of the DRAM algorithm.

Parameter Name	Value
Proposal through mean	True
Likelihood Scale	0.5
k_0	50
Rejection Scaling	0.1
s_0	2.4
Starting Sample	Spread
Boundary Handling	Ignore
Number of Chains	4
Burn-in Length	20%

5.3 Likelihood and Proposal Covariance Matrix

We begin again by testing the different likelihood functions and proposal covariance matrices. But this time we use only the scaled diagonal matrices based on the size between the boundaries since those matrices performed better in the MH section than using covariances from other algorithm runs, and because the DRAM algorithm updates and adapts its covariance matrix on its own to represent correlation between dimensions. The scaling factors we use for the initial covariance matrices are 0.5, 0.1, 0.05, 0.01, 0.005, and 0.001.

5.3.1 Evaluation

We first filter out those results whose box plots and histograms show no convergence in multiple dimensions, either because the samples do not move from the starting position or because they move uniformly across the whole possible space within the boundaries. The error functions support this decision, as most of the removed runs are in the worse half in regards to mean MAE values. In general, we exclude the filtered results from further analysis, unless specifically mentioned, such as during the ESS analysis.

Execution Time

The execution time varies significantly for these tests, from 3636s ($\approx$ 1h) to 7793s ($\approx$ 2h 10min). That is because a single iteration can unfold in essentially three different ways concerning the execution time:

- The first and second proposals are out of bounds, so we do not run the model. This is the fastest way, as we do not execute the model
- We accept the first proposal after we run the model once with it, or the first proposal is out of bounds, but the second proposal is within the boundaries. Either way, we run the model once in this scenario, making it the second fastest
- The first and second proposals are within the boundaries, but we reject the first proposal. Thus, we run the model twice in one iteration

Generally, we can say that a higher covariance leads to faster results, likely because this causes more proposals to be out of bounds then and we have to run the model less often because of the *ignore* method. Furthermore, a likelihood function that generally has a higher acceptance rate (like *scipy_pre_mean_log_gaussian* or *scipy_yearly_mean_log_gaussian*) is faster, presumably because the second stage proposal is evaluated less frequently if the first proposal was already accepted, which saves a model run. But overall, a low acceptance rate results in a shorter execution time, likely because of the *ignore* method.

Acceptance Rate

For the acceptance rates, we observe the expected behavior, namely that the acceptance rates are, in general, higher than for MH, and that the second stage proposals that deviate less from already accepted samples due to the decreased covariance matrix are more likely to be accepted than the first-stage proposals. However, the acceptance rates are still below 1% except for the *scipy_pre_mean_log_gaussian*, which reaches acceptance rates of 3%, and are thus still unsatisfactory since acceptance rates around 10% are desirable.

R-Convergence

The R-convergence values do not vary as much between different runs as they do during the MH test runs. We get the best result for *rank/fold-*$\hat{R}$ of 3.32 from using the diagonal proposal covariance matrix scaled down by 0.5 and the *scipy_per_month_log_gaussian* likelihood function. The worst value of 5.24 stems from using the 0.1 proposal covariance matrix and the *scipy_log_gaussian*. However, we discover no correlation between proposal covariance matrix or likelihood function and *rank/fold-*$\hat{R}$, as we achieve the best *rank/fold-*$\hat{R}$ value when using the highest proposal covariance matrix, but the second best when using the smallest covariance matrix. It is similar for the likelihood functions, where the

third-best result uses the *scipy_log_gaussian* function, as does the worst and third-worst. Furthermore, as the best value is still far higher than the requested value of 1.1, we treat the R-convergence values as inconclusive in this section.

ESS

Of the results remaining after we filtered them based on the histograms and box plots, only the combination of the 0.5 covariance matrix and the *AR_log_gaussian* function has a low enough max *esss_buithanh* to allow for an additional run with at least 8 samples per chain, as it has a max *esss_buithanh* of 966. There are also two results among the removed that allow for an additional run with decreased sample size: 0.001 covariance matrix and *scipy_per_month_log_gaussian* with a max *esss_buithanh* of 865, and 0.001 matrix and *scipy_log_gaussian* with a max *esss_buithanh* of 975. Of the remaining results, the best has a max *esss_buithanh* of 1026 and the worst a max *esss_buithanh* of 1536. Thus, notably, none have the worst case max *esss_buithanh* of 4000, contrary to some of the MH results.

The additional diagnoses run with decreased sample size yields no significant changes to the original diagnoses with 32000 samples: 36 samples remain, of which 20 are unique. The mode is the same, so there is no deviation for mode MAE or mode RMSE, and only a small change for mean MAE and mean RMSE (13.58 to 13.53, and 24.62 to 24.39). The *rank/fold-*$\hat{R}$ result is also only slightly smaller, from 3.63 to 3.53. This shows that roughly the same results can be represented with only 36 samples instead of 32000, attesting to the ESS truly being the effective size for the given samples, as no valuable information is lost.

Error Functions

The five best results according to mean MAE are listed in Table 5.2. The *scipy_log_gaussian* function seems to be the best likelihood function, as the three best results based on mean MAE values all use this likelihood function. Additionally, the other results using this likelihood function are among the better half of the results. The results are similar for the other error functions, as four runs using the *scipy_log_gaussian* likelihood function are in the better half, with two of the three best using this likelihood function.

Table 5.2: The Error function results for the 5 best results according to mean MAE.

Likelihood Function	Covariance Matrix	mean MAE	mode MAE	mean RMSE	mode RMSE
scipy_log_gaussian	0.5	11.36	10.43	17.84	16.94
scipy_log_gaussian	0.1	11.69	10.50	18.37	17.01
scipy_log_gaussian	0.01	11.85	11.29	18.84	17.70
scipy_post_mean_log_gaussian	0.1	11.91	12.82	18.51	19.21
scipy_monthly_mean_log_gaussian	0.005	11.94	10.61	18.76	17.05

Notably, the *AR_log_gaussian* likelihood function achieves the best error values on the

test dataset again, such as a test mean MAE of 11.74 and a test mean RMSE of 20.38 when combined with the 0.5 proposal covariance matrix. However, the results for the test dataset are again significantly better than the results for the training dataset, on which the same settings only achieve a mean MAE of 13.58 and a mean RMSE of 24.62. Furthermore, the settings from Table 5.2 achieve only slightly worse results on the test dataset than the *AR_log_gaussian* likelihood functions, with an average difference of 0.45 for test mean MAE and 0.86 for test mean RMSE. The *scipy_monthly_mean_log_gaussian* function combined with the 0.005 covariance matrix also achieves the worst test error values among the five settings from Table 5.2.

But, while the combination of the 0.1 proposal covariance matrix and *scipy_log_gaussian* function has the second-best mean MAE, it also has the worst *rank/fold-*$\hat{R}$ result, and is in the worse half of max *esss_buithanh* values with 1457. The 0.5 and 0.01 proposal covariance matrices with *scipy_log_gaussian* have good max *esss_buithanh* values with 1199 and 1237, making them the third and fourth best in the ESS diagnostics, respectively. For R-convergence values, the two runs are also close, with the 0.01 proposal covariance matrix having a *rank/fold-*$\hat{R}$ value of 3.44, and the 0.5 covariance matrix a value of 3.67, which makes the former the third best overall.

However, the 0.5 matrix generally leads to worse results, and, combined with the *scipy_log_gaussian* likelihood function, it has a lower acceptance rate of 0.24% than the 0.01 matrix of 0.58%. Thus, we use the diagonal matrix scaled down by 0.01 together with the *scipy_log_gaussian* likelihood function in the next sections, as this is among the best results in all relevant diagnostics.

5.4 Proposal Through Mean

In this section, we compare the two different methods we use to make second stage proposals: Either through the mean between the first proposal and the current sample, or using just the current sample as already described in Section 5.1.1. We name these methods *true* and *false* for convenience, respectively. As there are only two options, we do three algorithm runs per option as an exception to obtain more reliable results, as it is not too time-intensive. Furthermore, we decided to decrease the required ESS per chain from 8 to 6, as the previous requirement was evidently too strict. Thus, an ESSS of 1333 is now sufficient to do an additional run over the diagnostics with the current burn-in length.

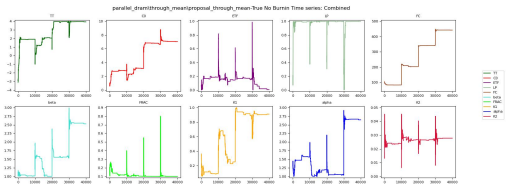
For the error function values, the *true* option is better since it has both a smaller average result over three runs, as can be seen in Table 5.3, as well as lower variance between the three results per setting. This difference is especially notable when using the mean of the samples to evaluate the model, making it more consistent. The same pattern persists for the test error values, although with a smaller difference. The *false* method has slightly better mean results for the ESS and R-convergence diagnostics with a max *esss_buithanh* of 1379 and a *rank/fold-*$\hat{R}$ of 3.77 compared to 1399 and 3.79 with

true. The former also has lower variance for *rank/fold-*$\hat{R}$ with 0.22 compared to the latter with 0.24, but a higher variance for max *esss_buithanh* with 153 compared to 60.

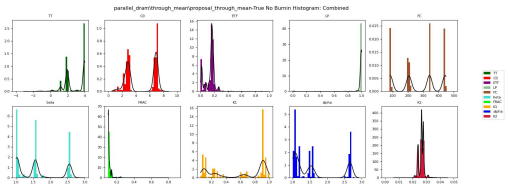
Table 5.3: The mean of the error function results.

Through mean	mean MAE	mode MAE	mean RMSE	mode RMSE
False	13.09	12.58	20.01	19.70
True	12.20	11.96	19.12	18.73

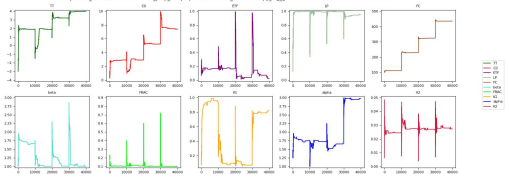
Interestingly, we notice in Figure 5.1 that the values of *beta* no longer converge as quickly as previously. However, we are not certain what caused this change, as the settings are the same for the three runs with *true* as for previous runs. Thus, the change likely stems from the randomness of MCMC algorithms. The third and especially the fourth chains – the chains furthest from the previously observed convergence area for *beta* – have the most problems with converging to that point. This might be because *LP* converges quickly towards the upper boundary for those chains, which causes more proposals to be out of bounds and be rejected. Consequently, the samples change less and converge more slowly.



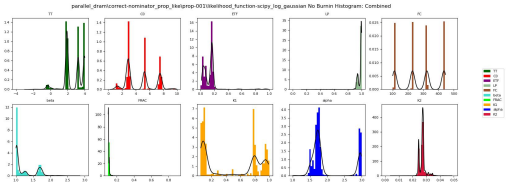
(a) Trace plot of bad convergence



(b) Histogram of bad convergence



(c) Trace plot of better convergence



(d) Histogram of better convergence

Figure 5.1: Trace plots and histograms of two runs with the same settings, showcasing bad and better convergence for *beta*.

Out of curiosity, we have also tested if it would change anything if we combine both options, i.e., make the proposals based on the mean but use the symmetric definition of the acceptance probability from Equation 5.2. However, the different acceptance probability definition would not have made a difference since we would have still accepted and rejected the same proposals. We assume this is because the values of the *scipy_log_gaussian* likelihood function are so sensitive and thus have large fluctuations with minute changes that the influence from the additional proposal distribution is insignificant.

Overall, we decide that the *true* option is better as it is more consistent according to our results, and performs remarkably better for the error functions while only being slightly worse in the ESS and R-convergence diagnostics. So, we continue to use the *true* option in the tests in the next sections.

5.5 Likelihood Scale

We test seven different values for σ as the scale value in the *scipy_log_gaussian* likelihood function: 20, 5, 2, 1, 0.5, 0.1, 0.05, of which we used 0.5 in the previous sections.

First of all, we again could not discover any correlation between the likelihood scale and the acceptance rate or execution time as using a likelihood scale of 2 resulted in the largest values in both categories, even though 2 is almost the median of our tested scale values. Since there is no clear best option for the likelihood scale, we describe each result consecutively in more detail to give a deeper understanding of our decision.

A scale value of 0.1 is the worst choice according to our tests, as it has the third worst *rank/fold-*$\hat{R}$ with 3.95, and the worst error functions values with a significant difference of roughly 4 when using the mode of samples and 3 when using the mean of the samples to evaluate the model compared to the second-worst result. Furthermore, it has a max *esss_buithanh* value of 1368, so its ESS per chain is barely below the required value of 6 and thus unable to do an additional diagnoses run with ESS.

Excluding the results from using 0.1 as the scale value, $\sigma = 1$ has the worst value for the error functions when using the mode of samples, but the best when using the mean. Additionally, using the ESS to reevaluate the diagnostics again, this setting has the second-best error function values using the mean (the best is still its evaluation with full sample size) and the best using the mode of samples. The max *esss_buithanh* is exactly at the upper boundary of 1333, at which the ESS per chain is 6. This might explain why there is such a large difference in the mode of samples between the ESS diagnostics run and the normal one, as most samples are removed with the ESS.

Scales of 2 and 0.05 have max *esss_buithanh* values of 1440 and 1498, so their ESS is too small to evaluate the diagnostics a second time. They also have the next worst error value when using the mode with 12.59 and 13.02 for mode MAE, and 19.01 and 19.86 for mode RMSE, respectively. Interestingly, the mean and mode error function values are very close to each other with a scale of 2, whereas for most other likelihood scales, the mode is smaller by roughly 1. Their mean error function and *rank/fold-*$\hat{R}$ values are only average, so we exclude these scale values as possible options as well.

All other likelihood scales result in additional diagnostic runs with ESS, where the mode is the same and the mean error values are slightly worse by up to 0.02, although the *rank/fold-*$\hat{R}$ value improves by 0.84 on average.

The result from using a likelihood scale of 20 has the second-best mode error values, but relatively bad mean errors with 12.41 for MAE and 19.62 for RMSE. This setting also results in the second highest *rank/fold-*$\hat{R}$ of 4.94.

Similarly to $\sigma = 2$, $\sigma = 5$ also results in a low difference between mode and mean

error functions (from 12.11 to 12.42 for MAE, and 18.99 to 19.62 for RMSE). However, these results are significantly worse than the best results, especially for the mode results, with a difference of almost 2 to the fifth best. It also only has an average *rank/fold-*$\hat{R}$ value, but it has the best *esss_buithanh* result with 1088 in this section.

Thus, scale values of 0.1, 5, and 20 are excluded from further consideration due to their high (mean) errors and R-convergence results. Similarly, scale values of 2 and 0.05 result in too high ESSS and are thus also ruled out. Our final decision is between a likelihood scale of 1 and 0.5. We proceed to use a scale of 0.5 since its error function values are more consistent (compared to a scale of 1 having the worst and best results), with only a small difference to the best: a difference of 0.16 for mode MAE, 0.43 for mean MAE, 0.39 for mode RMSE, and 0.89 for mean RMSE. A likelihood scale of 0.5 also results in a slightly better *esss_buithanh* with 1321 compared to 1333, and a significantly better *rank/fold-*$\hat{R}$ value of 3.6 compared to 5.17.

5.6 Non-Adaptation Length

In this section, we test different values for k_0, representing the non-adaptation length between updates of the covariance matrix. Specifically, we compare results between using a k_0 of 5, 50, 100, and 500, i.e., 2000, 200, 100, and 20 updates per chain in one run.

A k_0 of 5 has with 0.80% the highest acceptance rate and thus almost double the acceptance rate in the first stage compared to the next highest with $k_0 = 50$. This is expected, since the former option adapts the most among the tested settings to the obtained samples. The acceptance rate decreases with increasing k_0, although with only very minute differences of 0.0225% between $k_0 = 50$ and $k_0 = 100$, and 0.0125% between $k_0 = 100$ and $k_0 = 500$. But in everything else, $k_0 = 5$ has the worst results by a significant margin, such as approximately a difference of 4 for the mean RMSE, 3 for the mode RMSE, or 315 for max *esss_buithanh*.

The *rank/fold-*$\hat{R}$ values roughly decreases with increasing k_0: We achieve the best value at $k_0 = 500$ with 3.19, followed by $k_0 = 50$ with 3.47 as an outlier. But the remaining two values follow this pattern again, with $k_0 = 100$ with 3.70, and $k_0 = 5$ with 4.61, and thus a comparatively large gap to the next best.

The *esss_buithanh* values decrease with increasing k_0, ranging from 1550 for $k_0 = 5$ to 1235 for $k_0 = 50$ and 1135 for $k_0 = 100$ to 1116 for $k_0 = 500$. Thus, we have performed additional diagnostics runs with ESS for all settings but $k_0 = 5$. The mode error values only improve for the ESS run with $k_0 = 100$, and everything else stays almost the same with only slight variation in *rank/fold-*$\hat{R}$ values.

Excluding $k_0 = 5$ and including the results with reduced sample size, the other values have roughly the same error values when using the mean of samples, ranging from 12.63 to 12.07 for mean MAE and 19.55 to 18.77 for mean RMSE. The mean MAE values improve with increasing k_0, the mean RMSE values almost follow the same pattern, although $k_0 = 100$ is slightly better with 18.77 than $k_0 = 500$ with 18.80. For the mode error values, we discover no pattern as the ESS run with $k_0 = 100$ has the best values

with 10.51 and 16.90, $k_0 = 50$ is the next best with a difference of 0.3 and 0.17, and $k_0 = 500$ is quite far behind with 13.89 and 20.86 for MAE and RMSE.

Thus, we use $k_0 = 500$ in the next sections as it has the best *rank/fold-*$\hat{R}$ value, the best max *esss_buithanh*, and the best mean MAE as well as the second best mean RMSE. While this setting has bad results when using the mode of samples for the error functions, we still use it due to its performance in the other diagnostics and because the mean error values are more representative and thus reliable than the mode errors.

5.7 Rejection Scaling

Here, we discuss the results from using different factors when scaling the proposal covariance matrix for the second stage. We test values of 0.01, 0.1, 0.5, 1, and 2, i.e., three values that decrease the covariance, one that does not change it, and one that increases the covariance matrix.

Acceptance Rate

Unsurprisingly, a scaling factor of 2.0 has the worst acceptance rate, as the second stage acceptance rate is with 0.08% half of the rate when using a scaling factor of 1, which is the next worst acceptance rate. In general, the acceptance rate increases with decreasing rejection scale, as described in Section 5.1.1.

Error Functions

A scaling factor of 0.1 results in the worst mode error values with 13.14 for MAE and 19.69 for RMSE, whereas 0.01 has the best mode error values with 10.79 and 17.01. The other settings are within a difference of roughly 1 to the best, ranging up to 11.37 for MAE and 17.90 for RMSE.

However, 0.01 has the worst mean error values, though only with a difference of 0.28 for MAE and 0.63 for RMSE from the best result. So, the mean error values are all quite close in this test round, and thus inconclusive.

MCSE

Interestingly, the mean MCSE values are all close to 0.15 except for a scaling factor of 1, which has a mean MCSE of 0.56. This might be because of its larger covariance and only slightly smaller acceptance rate in the second stage than the other settings with smaller scaling factors. This scaling factor also results in a high acceptance rate for the first stage. Combined, this leads to an overall higher number of unique samples after burn-in.

R-Convergence

We can essentially organize the *rank/fold-*$\hat{R}$ results into two groups: One group contains the results with scaling factors 0.01, 0.1, and 1, as their *rank/fold-*$\hat{R}$ value is close to 3.75.

The second group contains the factors 0.5 and 2, as their values are at 4.28 and 4.43. Thus, the members of a group are close to each other and quite distant from the other group.

ESS

The max *esss_buithanh* values have their minimum at the scaling factor of 0.5 with 1244, and then increase both with increasing and decreasing scaling factors to 1485 with a factor of 0.01, and to 1382 with a factor of 2. However, only the scaling factor of 0.5 achieves a low enough correlation to do an additional diagnostics run with ESS, though this barely changes the results, with only *rank/fold-*$\hat{R}$ increasing by 0.3.

Thus, 0.01 and 0.5 seem like the best choices for the scaling factor of the covariance matrix of the second stage. Consequently, it is the decision between the best mode error values and second best mode error values, between the worst max *esss_buithanh* and the best max *esss_buithanh* value, and between an average *rank/fold-*$\hat{R}$ and the second worst *rank/fold-*$\hat{R}$ value. Therefore, we use both in the next section for further evaluation.

5.8 AM Scaling Factor

We test the values of 1.2, 2.4, and 3 for s_0, that is, values of 0.144, 0.576, and 0.9 for s_d. As there are only three values, we have performed two runs each with the same value to obtain more reliable results. Combined with the two settings for the rejection scaling factor, we have 12 different runs.

5.8.1 Rejection Scaling Factor

But first, we discuss the result between the rejection scaling factor of 0.01 and 0.5, which we obtained while testing the different AM scaling factors. The latter is better in our opinion as their error values are all smaller by roughly 0.2, and they result in a lower variance than using a scaling factor of 0.01, except for the mean RMSE.

This scaling factor also has a smaller average max *esss_buithanh* with 1367 compared to 1454. Thus, it has allowed us to do two additional diagnoses with ESS over the six executions, whereas the scaling factor of 0.01 achieves only one. However, the latter's variance for max *esss_buithanh* is with a value of 54 lower compared to 0.5 with 116. But, since the smallest and largest max *esss_buithanh* are both smaller with a factor of 0.5 than with 0.01, 0.5 is also better in the ESS diagnostics in our opinion.

Only for the *rank/fold-*$\hat{R}$ is the scaling factor of 0.01 slightly better, with a value of 4.01 and a variance of 0.48 compared to 4.03 with a variance of 0.64.

So, since 0.5 achieves overall better results, we use a value of 0.5 as the scaling factor for the second stage covariance matrix, and only focus on the results obtained with this setting to evaluate the AM scaling factors.

5.8.2 AM Scaling Factor, s_0

R-Convergence

For the *rank/fold-*$\hat{R}$ values, $s_0 = 1.2$ performs the worst as it has both the best and worst value with 3.49 and 5.62, and therefore seems unstable and unreliable. Furthermore, $s_0 = 2.4$ obtains values of 3.51 and 3.50 in its two runs, which makes it almost as good as 1.2's best run and far more reliable. Lastly, $s_0 = 3$ has values of 4.2 and 3.88 and is thus more consistent than $s_0 = 1.2$, but still comparatively far behind $s_0 = 2.4$.

ESS

We observe a similar pattern for the max *esss_buithanh* values: $s_0 = 1.2$ has the worst result with an average of 1477 and is thus not able to do an additional diagnostics run with ESS; $s_0 = 2.4$ has the best average of 1222; and $s_0 = 3$ is again the median with an average of 1402. But both $s_0 = 2.4$ and $s_0 = 3$ have one run with a max *esss_buithanh* below 1333 – 1060 and 1324 to be precise – and thus allow for an additional diagnostic run. Those additional diagnostics result in almost the same values, except for *rank/fold-*$\hat{R}$ where $s_0 = 2.4$ improves to 3.19 and $s_0 = 3$ worsens to 4.80.

Error functions

However, for the mode error values, $s_0 = 3$ performs far better with a MAE result of 10.65 and RMSE of 16.95 compared to $s_0 = 2.4$ with a MAE of 12.63 and RMSE of 19.99, and $s_0 = 1.2$ with a MAE of 13.08 and RMSE of 19.71. The mean error values are again not decisive as they are all very close together, since $s_0 = 1.2$ has the worst results, but is behind $s_0 = 3$, the best option, by only 0.16 for MAE and 0.13 for RMSE.

Thus, our decision lies between $s_0 = 2.4$ with the best results in all but mode error values, and $s_0 = 3$, which has the best mode error values, but average to bad results in the other diagnostics. Therefore, we continue to use $s_0 = 2.4$, as we believe its advantages are superior to its one disadvantage we discovered. Especially, since this disadvantage is the bad results for the mode errors, which are less indicative of the overall result than the mean errors.

5.9 Starting Sample

We use the same options for selecting starting samples as in Section 4.6.1, although we use the mean and mode values of samples we have obtained during this chapter so far, as listed in Table 5.4. However, this time, the mode before and after burn-in is the same, so we do not differentiate between them. Naturally, the mean values vary between with and without burn-in, but only slightly, and thus do not warrant an additional case. We also see that the mean and mode are fairly close for most dimensions, which suggests better convergence than for the MH algorithm.

Table 5.4: The mean and mode for each dimension based on the previous samples after burn-in.

Name	Mean Burn-in	Mode
TT	1.97	-2.25
C0	4.55	3.86
ETF	0.16	0.17
LP	0.94	0.99
FC	272.54	233.65
beta	1.45	1.00
FRAC	0.18	0.15
K1	0.47	0.25
alpha	1.86	1.67
K2	0.024	0.007

Error functions

The error functions do not give decisive information, as most have similar values. The exceptions are the worst result using *random*, which differs by at least 1 from the second-worst for all error functions, and the second-worst using *spread*, which differs by more than 1 from the third-worst for the mode errors.

R-Convergence

As for the *rank/fold-*$\hat{R}$ value, *default* has by far the worst value with 5.36, followed by *spread* with 4.39. The other options result in *rank/fold-*$\hat{R}$ values between 3.94 with *close mode* and 3.51 with *close mean*.

ESS

Only *random*, *close mode*, and *close mean* have a max *esss_buithanh* below 1333 with 1261, 1225, and 1295, respectively, and are thus allowed to do an additional evaluation with reduced sample size. However, *spread* and *close default* barely have a too high value with 1347 and 1356. The other options result in max *esss_buithanh* values above 1400, with *default* being the worst with 1539. The additional ESS runs improve the *rank/fold-*$\hat{R}$ values to 3.19 for *close mean* and 3.16 for *close mode*, making them the second best and best overall, and worsen *random* to 4.30. The other diagnostics barely change with reduced sample size.

Therefore, we use *close mean* as our method to determine the starting samples since it results in the third best max *esss_buithanh* value with only a difference of 70 to the best, and the best *rank/fold-*$\hat{R}$ with 3.51 when excluding ESS runs. Including the additional

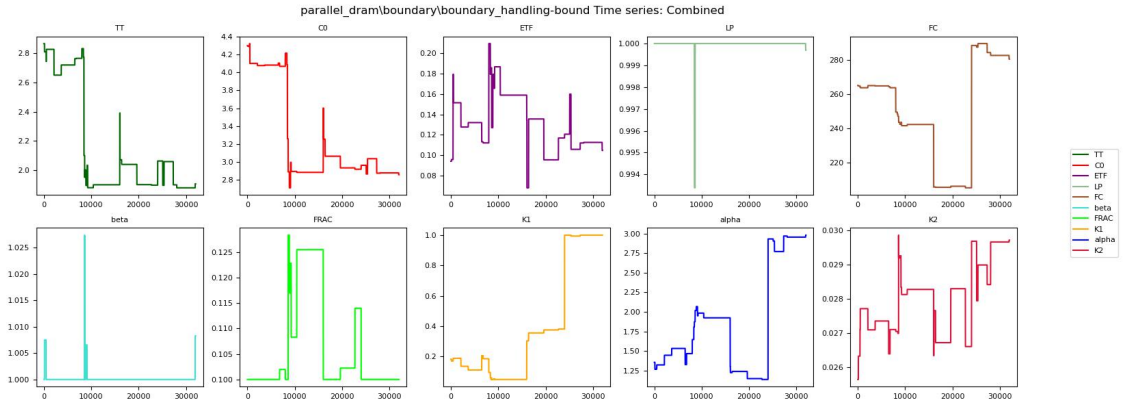


Figure 5.2: Trace plot of the model run using *bound* to handle out of bounds samples

evaluations, its ESS run is the second best *rank/fold-*$\hat{R}$ and its evaluation with full sample size is the third best overall. Although *close mode* also leads to good results, it does not matter much as *close mode*'s and *close mean*'s base values are quite similar except for *TT*, so they propose similar starting samples anyway.

5.10 Boundary Handling

Most boundary handling methods work the same as we described in Section 4.7. Only *ignore* works differently, as we might have to use an out-of-bounds first proposal z' to calculate the acceptance rate for a second proposal z'' lying within the boundaries. In this case, $\alpha_1(z' \mid z'')$ and $\alpha_1(z' \mid z_k)$ in Equation 5.2 are 0 as z' has a likelihood of 0. However, we evaluate the proposal distributions such as $q_1(z' \mid z'')$ as normal, i.e., with z' still out of bounds.

As for the MH algorithm, the most significant difference we observe is that *ignore* is almost 4 times faster than the other methods, which all have a similar execution time. The only other noticeable difference is that *bound* has the worst possible max *esss_buithanh* of 4000. This is because most chains stay at the boundary value for *LP* and *beta*, as we can see in Figure 5.2. Since their posterior's mode seems to be near the boundary, many proposals are out of bounds, which are mapped to the boundary value through the *bound* method. Therefore, the chains are mostly stuck at the boundary value in these dimensions. The other options result in a max *esss_buithanh* between 1402 with *eflect* and 1454 for *fold*. Besides *bound* being the worst for the ESS values, *fold* has the worst results for everything else, although only by a small difference. For example, the mode error values differ by less than 1.0, the mean error values by at most 0.4, and the *rank/fold-*$\hat{R}$ values by 0.2.

Examining the error values more closely, we observe that *ignore* has the second-worst error values except for the mean RMSE, where it has the best result and is thus very slightly ahead of *eflect*. For the MAE values, *eflect* and *bound* have a small difference

below 0.01, and a slightly larger difference of 0.12 for the mode RMSE; *eflect* performs better for the mean values, whereas *bound* has better mode results. However, all methods achieve relatively close error values, with differences below 1 for mode MAE and mode RMSE, and 0.4 for mean MAE and mean RMSE. This means the different methods achieve similar results overall, since the mean over samples is more representative of the entire run than the mode. The error values on the test dataset are similarly close, although *eflect* is the best in all functions except for test mean RMSE, where *ignore* is the best. The *ignore* method is also the second-best in the other three test error functions, with a maximum difference of 0.1.

Thus, we decide to continue to use *ignore* to ensure the model is not run with out-of-bounds samples, mostly because of its significant execution time advantage. Additionally, this method still results in the second-best *rank/fold-*$\hat{R}$ value with 3.97 to *bound*'s 3.91, and the second-best max *esss_buithanh* with 1446. It performs slightly worse for most error values, but the differences are only 0.4 for the mode values, making it the third-best option in this regard, 0.08 for mean MAE, and it is the best option for mean RMSE by 0.001.

5.11 Number of Chains

As the next option, we again test how using different numbers of chains influences the results. We test the values 2, 4, 6, and 8 for M as in the previous chapter.

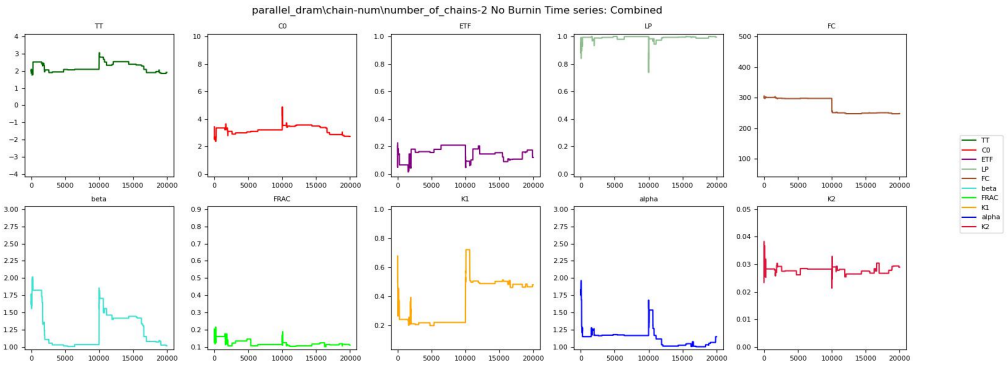
We observe that the acceptance rate is almost constant for all options at around 0.25%, and consequently that the number of unique samples and the execution time increases with the number of chains.

Using 2 chains results in the lowest error values overall, whereas 4 chains have the worst mean error values, and 6 the worst mode error values. However, the difference between best and worst is only 0.5 and thus negligible. It is similar for the test dataset, although 4 chains slightly outperform 2 chains for the test mean MAE and test mean RMSE.

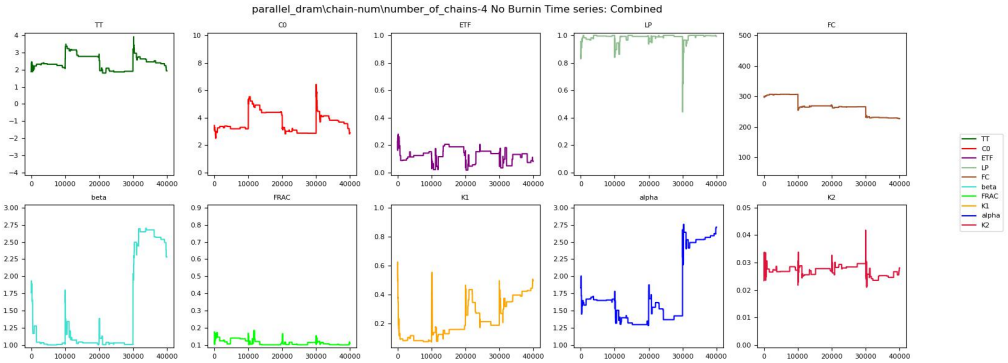
Similarly, 2 chains have by far the best *rank/fold-*$\hat{R}$ value with 1.94, whereas the others have values around 3.3, besides 8 chains with 5.23. However, fewer chains make the R-convergence result less reliable, as fewer chains have to show the same behavior. Furthermore, as shown in Figure 5.3, the starting positions in the run with 2 chains are closer to each other than in the run with 4 chains, further facilitating low R-convergence values.

Utilizing 4 chains results in the best max *esss_buithanh* value with 1043, whereas the others have values at 1474 with 2 chains, or above, and are thus significantly worse.

Since the *rank/fold-*$\hat{R}$ result for 2 chains is unreliable, and because 4 chains show good results, especially for the ESS values, and have a decent execution time, we continue to use 4 chains. Furthermore, the result with 4 chains has slightly smaller test mean error values, so it generalizes moderately better than 2 chains.



(a) 2 chains



(b) 4 chains

Figure 5.3: Trace plots of two runs using 2 and 4 chains.

5.12 Burn-in Length

For the burn-in length, we test the same factors as before, i.e., 0%, 20%, 50%, and 80%. As the burn-in length does not affect the algorithm run itself, we reuse the result from the previous section and change the number of samples we remove and keep.

The most noticeable result is that the mode error values are the same for all burn-in lengths besides 80%, which is better by about 0.2 for both MAE and RMSE, with 11.10 and 17.20. The mean values show no pattern as a length of 20% has the best MAE result with 11.57 and a length of 50% the best for RMSE with 18.02, whereas we obtain the worst results with a burn-in length of 0% with 11.69 for MAE and 18.24 for RMSE. So, the error values are very similar again for each option.

The *rank/fold-*$\hat{R}$ values also increase as we discard more samples, just like for the MH algorithm. However, the increase is not as drastic as for the MH algorithm, since a length of 0% has a *rank/fold-*$\hat{R}$ value of 3.28 and a length of 80% results in a value of 4.77, compared to the value of 16.76 we have with the MH algorithm.

Since one chain accepted no new samples in the last $\approx$ 3000 iterations, the max *esss_buithanh* for a burn-in length of 80%, i.e., when keeping the last 2000 samples, is the worst case of 1001. Excluding this, the ESSS decreases as we discard more samples, from 1321 to 1043 to 834. But since the sample size also decreases, a burn-in length of 20% has the largest ESS with 7.67 per chain, slightly ahead of a length of 0% with 7.57. But since the ESS is rounded down to the smaller integer, they are essentially the same. The burn-in length of 50% barely misses the required ESS of 6 with a value of 5.995, so that we only perform additional diagnostic runs with ESS for burn-in lengths of 0% and 20%. This additional evaluation improves the *rank/fold-*$\hat{R}$ value by about 0.2 and 0.3, respectively, and slightly worsens the mean error values for the former case.

Thus, we continue to use a burn-in length of 20% as it results in the best max *esss_buithanh*, has the second best *rank/fold-*$\hat{R}$ value with 3.60, and the best (for mean MAE) or almost best error values.

5.13 Parallel DRAM against Basic DRAM

As with the MH algorithm, we also perform one execution of the Basic DRAM algorithm with only one chain.

The error values show similar behavior as with the MH algorithm, but with smaller differences, that is, slightly worse mode values by about 0.08 for both MAE and RMSE, but a slightly larger improvement for mean values of 0.22 for MAE and 0.5 for RMSE. Since it is only a single chain, the mean, min, and max *esss_buithanh* are all the same with 872, and thus are similar to the mean *esss_buithanh* with 897 of the run with 4 chains, but have a better max. Thus, we can reevaluate the diagnostics with ESS, although this only includes the error functions and MCSE, since we have only one chain. However, this run achieves very similar results to the original run.

As expected from the results in the previous chapter and Section 5.11, there is no

significant difference between using basic DRAM or our simple parallel DRAM version. Thus, we continue to use the parallel version with 4 chains for the same reasons mentioned in Section 4.10

5.14 Summary

The best parameters for the DRAM algorithm according to our tests are summarized again in Table 5.5. The results we obtained with those parameters are listed in Table 5.6 and Figure 5.4. For the single-chain autocorrelation, we again selected only one chain to save space, and because this time, there is no significant difference between chains.

Table 5.5: The final parameters of the DRAM algorithm.

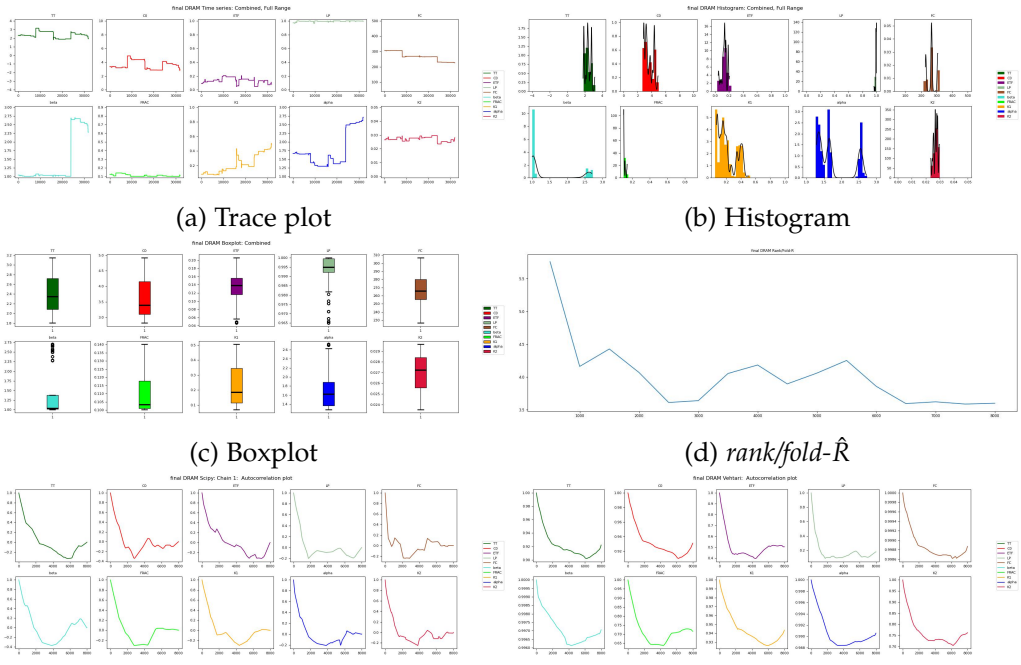
Parameter Name	Value
Proposal Covariance Matrix	0.01
Likelihood Function	*scipy_log_gaussian*
Proposal through mean	True
Likelihood Scale	0.5
k_0	500
Rejection Scaling	0.5
s_0	2.4
Starting Sample	*close mean*
Boundary Handling	Ignore
Number of Chains	4
Burn-in Length	20%

We can see that the DRAM algorithm has less variance among different settings and generally better results than the MH algorithm, e.g., in the *rank/fold-*$\hat{R}$ value with different burn-in lengths. Additionally, we see in Table 5.4 and Table 4.6 that the former algorithm's mean and mode over all samples are closer to each other than the latter algorithm's. Additionally, by far the most significant parameters are again the proposal covariance matrix and the likelihood function.

However, there is no significant improvement for the error functions compared to the MH algorithm, and the diagnostics with multiple chains are still not usable (ESS) or do not show satisfactory results (R-convergence). But, since we only use a simple method to create multiple chains at once, this is not surprising.

Table 5.6: The final results using the mean and mode of samples obtained through the settings in Table 5.5.

Name	Mean	Mode
TT	2.37	2.77
C0	3.61	4.39
ETF	0.14	0.19
LP	0.99	0.99
FC	267.41	268.70
beta	1.42	1.03
FRAC	0.11	0.10
K1	0.21	0.16
alpha	1.74	1.30
K2	0.027	0.028
MAE	11.57	10.30
RMSE	18.07	17.45
mean *esss_buithanh*	897	
rank/fold-$\hat{R}$	3.60	
mean *split-*$\hat{R}$	9.21	



(a) Trace plot

(b) Histogram

(c) Boxplot

(d) *rank/fold-*$\hat{R}$

(e) Single chain autocorrelation

(f) Multiple chains autocorrelation

Figure 5.4: Diagrams for the final DRAM run.

6 Differential Evolution Adaptive Metropolis algorithm

The **D**iffe**R**ential **E**volution **A**daptive **M**etropolis (DREAM) algorithm is another MCMC algorithm that is even more advanced than the DRAM algorithm. It is the last algorithm we discuss and test in this paper, and the first that naturally uses multiple chains. It requires at least three chains to function. It is also not reliant on a proposal distribution, which, as we have seen, can strongly influence the performance of the previous algorithms. [5]

As there is already a detailed, public implementation of the DREAM algorithm in Python, called PyDREAM[1], we use this version. This version implements MT-DREAM$_{ZS}$, a subversion of DREAM. But since we do not enable multitry (MT) for time efficiency reasons, as it would otherwise run the model multiple times in a single iteration and thus require more than ten times its current execution time, we essentially use DREAM$_{ZS}$. We explain the relevant difference for us between DREAM and DREAM$_{ZS}$ in Section 6.10. However, we first describe the general approach of the DREAM algorithm.

6.1 Introduction

Before we explain how the DREAM algorithm works, we begin by explaining the idea behind the Differential Evolution Markov chain (DE-MC) algorithm, as the DREAM algorithm expands and improves upon it.

6.1.1 DE-MC

For the DE-MC algorithm, at each iteration k, the current state $z_{k,m}$ of all chains defines a population Z, which is an $M \times d$ matrix [5]. Thus, Z_m refers to the current state of chain m. Instead of obtaining a new proposal through a proposal distribution, we create it based on the states of other chains:

To create a proposal for the chain m at iteration k, we first select two distinct integers a and b from $\{1, \ldots, m-1, m+1, \ldots, M\}$, which define the other chains we use to make the new proposal for chain m. Then, the new proposal z'_m is created as

$$z'_m = z_{k,m} + \gamma_d(Z_a - Z_b) + e^{-6} * \zeta_d, \tag{6.1}$$

where ζ_d is a d-dimensional vector containing values sampled from a univariate Gaussian with mean 0 and variance 1 to add some randomness, and γ_d defines the jump rate, i.e.,

[1] https://github.com/LoLab-MSM/PyDREAM

how much influence the other chains have. Typically, γ_d is chosen randomly in each iteration from $\{1, \gamma_0\}$ with a probability of 0.1 for $\gamma_d = 1$. So, γ_0 is the default jump rate, commonly selected as $2.38/\sqrt{2d}$. But with a low probability, $\gamma_d = 1$, and the chains a and b are fully integrated into the new proposal. [5, 17, 18]

Besides the proposal being created differently, the DE-MC algorithm works the same as the MH algorithm, so it accepts the proposal if $\alpha(z'_m \mid z_{k,m})$, defined as

$$\alpha(z'_m \mid z_{k,m}) = \min\left\{1, \frac{p(z')}{p(z_k)}\right\}, \tag{6.2}$$

is larger than a random value drawn from a uniform distribution ranging from 0 to 1. The chains do not have to be synchronized, as the population Z stores the current state for every chain. For example, the k-th iteration for chain m can occur and use Z_a even when chain a has performed fewer or more than k iterations. [5]

6.1.2 Introduction to DREAM

DREAM makes some changes to the DE-MC algorithm. One of the changes is that not all dimensions are updated even if a new proposal is accepted. We determine the dimensions to update through the crossover values: The crossover values CR are a geometric sequence over n_{CR} values, $\left\{\frac{1}{n_{CR}}, \frac{2}{n_{CR}}, \ldots, 1\right\}$, where each element has its individual selection probability. These probabilities are initialized with the same probability $\frac{1}{n_{CR}}$, and can be updated during the algorithm. After selecting a value cr from CR, we draw one value per dimension from a uniform distribution ranging from 0 to 1. If the value is below cr, the corresponding dimension is added to a set A and will be updated in the current proposal, otherwise, it stays the same. The size of A is d^*, so that $d^* \leq d$. Thus, $cr = 1$ updates all dimensions like the previous algorithms we discussed. If no uniformly sampled value is below cr, we select one dimension randomly or the dimension corresponding to the smallest sampled value. This way, we guarantee that d^* is at least one and thus at least one dimension changes in every proposal. We define the values of a sample with only the selected dimensions as $z^{=A}$, and the samples excluding those dimensions as $z^{\neq A}$. [5]

Consequently, Equation 6.1 is adapted to

$$\begin{aligned} z_m^{'=A} &= z_{k,m}^{=A} + (1_{d^*} + \lambda_{d^*})\gamma_{(\delta,d^*)} \sum_{j=1}^{\delta} (Z_{a_j}^{=A} - Z_{b_j}^{=A}) + e^{-6} * \zeta_{d^*}, \\ z_m^{'\neq A} &= z_m^{\neq A}, \end{aligned} \tag{6.3}$$

which already includes additional improvements. For example, δ describes how many pairs of chains are used for the single proposal, which is randomly selected each iteration from $\{1, 2, ..., \delta_{max}\}$. So, for DE-MC, δ_{max} is essentially 1 as it always uses only a single pair of chains. This is incorporated into $\gamma_{(\delta,0)}$ as it is now typically defined as $2.38/\sqrt{2\delta d^*}$, and the probability for $\gamma_{\delta,d^*} = 1$ is increased to 0.2. δ also influences

the minimum number of chains required since M must be at least $2\delta + 1$. However, for small d, $\delta = 1$ is fine, so we can keep using 4 chains. [5]

Additionally, 1_{d^*} represents a d^*-dimensional identity matrix, and λ_{d^*} is a value sampled from a uniform distribution centered on 0 with a small interval added for further randomness. [5]

DREAM can contain more features, such as outlier detection and correction for when a single chain is stuck and thus has a negative influence on the other chains. But the use of crossover values to update only some dimensions and using multiple chain pairs for one proposal are the major and defining changes of DREAM. So, we do not describe these additional features further. [5]

6.2 Initial Settings

In Table 6.1 we again list the initial parameter values we use until we determine a fitting value by testing them, except for the likelihood function, which we test first. For the DREAM-specific parameters, we use the standard values of PyDREAM except for the probability of $\gamma_d = 1$ and the snooker probability. We initially set the latter to 0 to disable snooker updates to make it more similar to the normal DREAM algorithm instead of DREAM$_{ZS}$ at first. We also set the length of crossover burn-in, i.e., the length during which the crossover probabilities are adjusted, to 0, which essentially disables the adaptation.

The execution time stays roughly constant at around 2h 30min for most options, so we do not mention it in each section. We also do not discuss the results of the additional diagnostics run with ESS, unlike in the previous section, since we already showed that there is no significant difference between the original evaluation and the evaluation with ESS. Furthermore, we set the minimum ESS per chain back to 8 as we expect better results with this algorithm.

Table 6.1: The other starting parameters of the MH algorithm.

Parameter Name	Value
Likelihood Scale	0.5
Starting Sample	Spread
Boundary Handling	Original (Eflect)
$P(\gamma_d = 1)$	0%
crossover_burn-in	0%
n_{CR}	3
δ	1
snooker probability	0%
Number of Chains	4
Burn-in Length	20%

6.3 Likelihood Functions

As the DREAM algorithm does not require a proposal covariance matrix, we have to test each likelihood function only once. We use the same likelihood functions as discussed in Section 4.4.1.

Acceptance Rate

The *scipy_pre_mean_log_gaussian* likelihood function results in by far the highest acceptance rate with 43.09%, followed by *scipy_yearly_mean_log_gaussian* with 12.26% and *scipy_post_mean_log_gaussian* with 11.18%. The other likelihood functions result in acceptance rates below 10%, with *scipy_log_gaussian* being the worst at 1.50%, though this is still higher than most settings we tried for the MH or DRAM algorithm. This mirrors the results from Table 4.2.

Error Functions

The *scipy_pre_mean_log_gaussian* and *scipy_yearly_mean_log_gaussian* functions have the worst performance for the error values, with values between 30 and 40 for MAE and 40 to 80 for RMSE. Especially, the mean RMSE values are bad, as both likelihood functions obtain values above 80 for them. This is because of their too large acceptance rate, which turns the posterior into a uniform distribution for some dimensions, as can be seen in the histograms in Figure 6.1. There, we see that the histograms and KDEs for *FRAC*, *K1*, *alpha*, and *K2* are almost flat, so that they give an equal probability for every value and thus no new information over the prior.

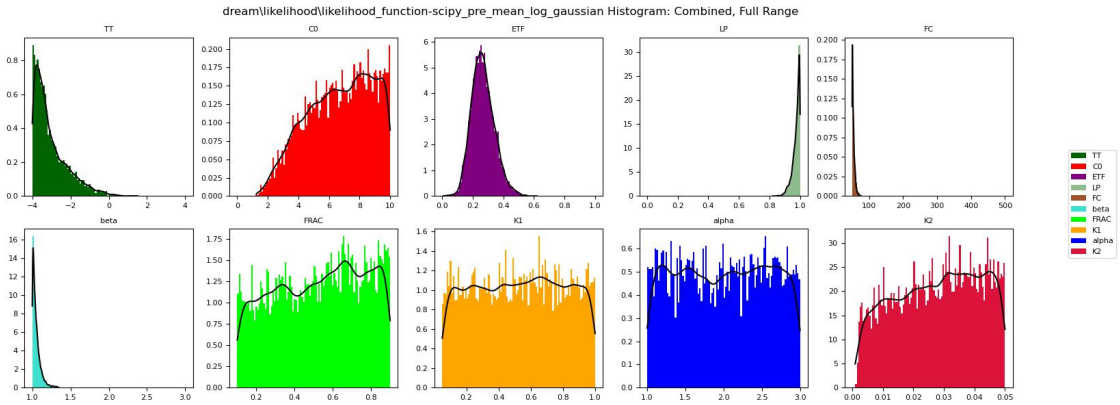


Figure 6.1: Histogram showing useless results in some dimensions due to high acceptance rate when using the *scipy_pre_mean_log_gaussian* likelihood function.

The error values of the four best likelihood functions according to mean MAE are listed in Table 6.2. The other likelihood functions have MAE values above 15 using the mode and above 14 with the mean, and above 25 for the RMSE regardless of using the

mean or mode. As we expect, *scipy_log_gaussian* has the best error values for all four diagnostics, although *scipy_post_mean_log_gaussian* is very close, especially for mean MAE. *scipy_monthly_mean_log_gaussian* has the third-best error values, and all three likelihood functions result in test MAE values around 12.

Table 6.2: The Error function results for the 4 best results according to mean MAE.

Likelihood Function	mean MAE	mode MAE	mean RMSE	mode RMSE
scipy_log_gaussian	10.15	10.15	16.523	16.521
scipy_post_mean_log_gaussian	10.19	10.34	16.58	16.94
scipy_monthly_mean_log_gaussian	11.18	10.96	18.34	17.98
scipy_per_month_log_gaussian	12.46	11.49	21.20	18.27

The same pattern holds for the RMSE values, where *scipy_log_gaussian* is the best and has barely any difference between mean RMSE and mode RMSE, followed closely by *scipy_post_mean_log_gaussian*, especially for the mean RMSE. The third-best likelihood function concerning the RMSE values is *scipy_monthly_mean_log_gaussian*, with a comparatively large gap to the best two.

MCSE

The *scipy_post_mean_log_gaussian* function has a surprisingly high mean MCSE of 1.25, followed by *scipy_pre_mean_log_gaussian* with 0.81. The latter we expect to have a high MCSE due to its high acceptance rate. The other functions result in mean MCSE values around or below 0.2. Similarly, for mean batch MCSE, the *scipy_post_mean_log_gaussian* has the second highest value with 303.89, with *scipy_monthly_mean_log_gaussian* having the highest with 365.52. The other functions obtain values around 200. The comparatively large values for *scipy_post_mean_log_gaussian* are perhaps because of its less peaked, but instead wider histogram for *alpha*, as shown in Figure 6.2. We also see that *FC* changes and is no longer stuck at its starting position as with the previous two algorithms.

R-Convergence

In this diagnostic, we see a substantial improvement over the previous two algorithms: Even the worst *rank/fold-*$\hat{R}$ values with 1.9 for *scipy_monthly_mean_log_gaussian* and 1.12 for *scipy_log_gaussian* are significantly better than what the two algorithms achieved. The other functions result in values at or below 1.04, so they converged according to the diagnostic. But as discussed in Section 4.4.3, the *scipy_pre_mean_log_gaussian* has not truly converged but only tricked the diagnostic with its large acceptance rate, as can be seen in Figure 6.1. Figure 6.3 is an example of average *rank/fold-*$\hat{R}$ progression, and Figure 6.4 is an example of comparatively bad progression. The important difference between these two figures is that the bad progression always has larger values than the average progression.

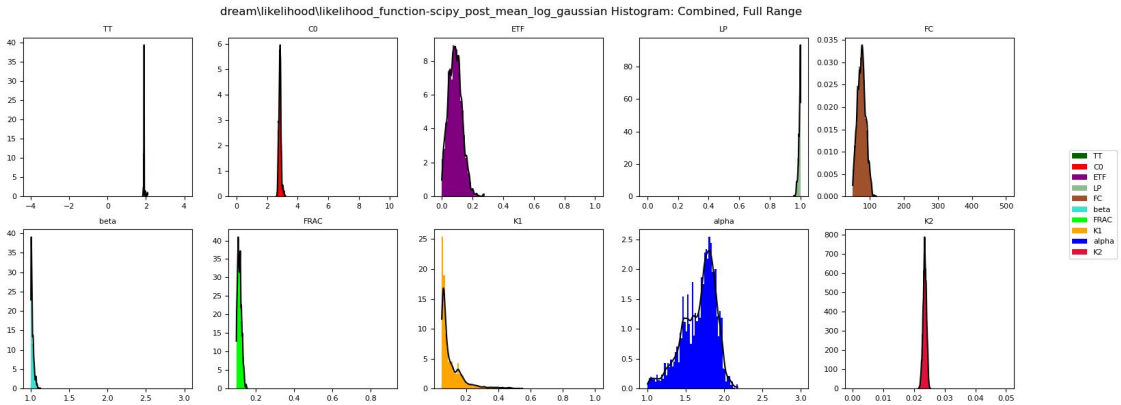


Figure 6.2: Histogram for a run using *scipy_post_mean_log_gaussian*.

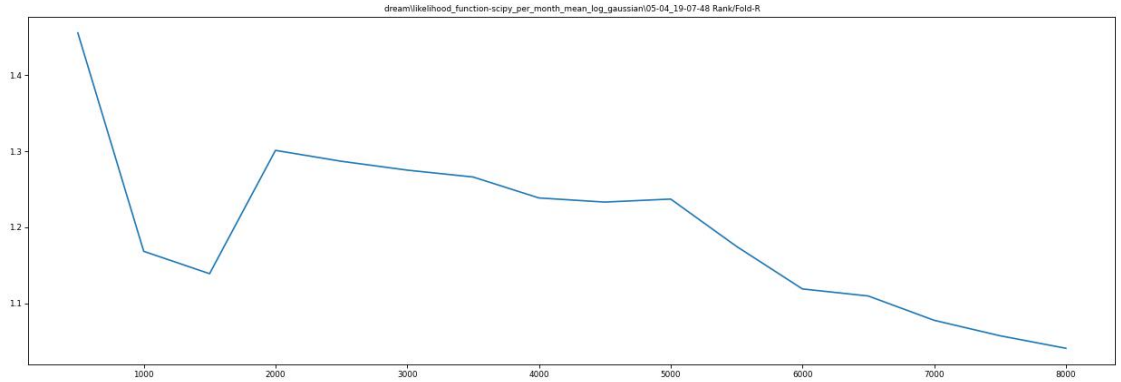


Figure 6.3: Average *rank/fold-*$\hat{R}$ progression with *scipy_per_month_mean_log_gaussian*

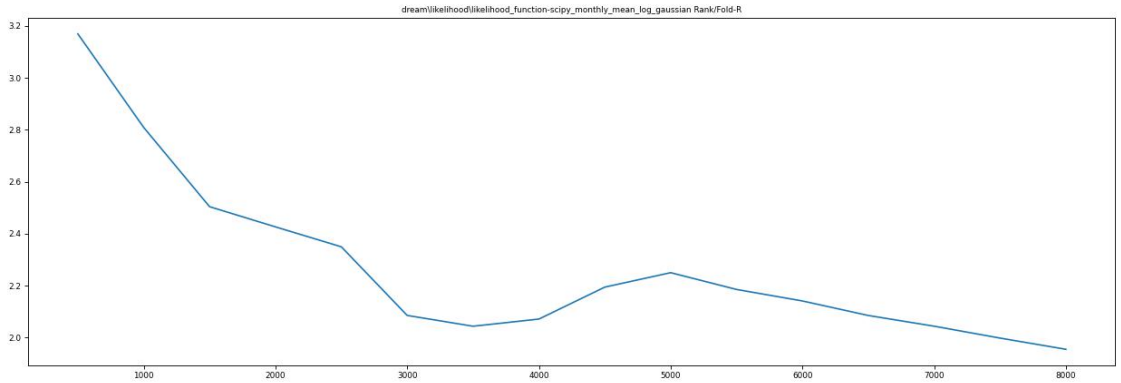


Figure 6.4: Bad *rank/fold-*$\hat{R}$ progression with *scipy_monthly_mean_log_gaussian*

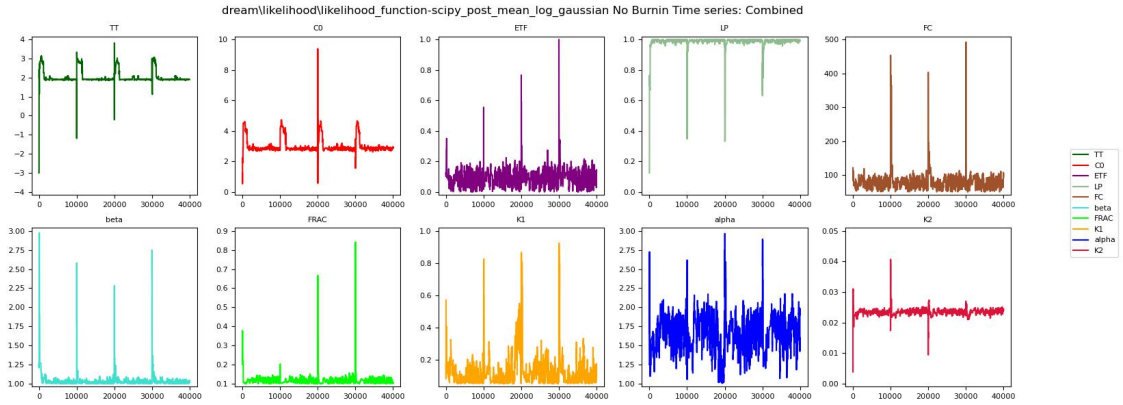


Figure 6.5: Trace plot of the run using the *scipy_post_mean_log_gaussian* function without burnin

The trace plot over all samples before the burn-in phase in Figure 6.5 also demonstrates how quickly the chains converge to similar values.

ESS

The most noticeable result for the ESS diagnostics is that we can calculate a bulk-ESS value for every option besides *scipy_monthly_mean_log_gaussian*. The autocorrelation plot for this function is in Figure 6.6. It resembles the autocorrelation plots from the MH and DRAM results significantly for *TT* and *C0*, which also could not calculate bulk-ESS values. This likelihood function is also the only one that does not result in an additional diagnostics evaluation with ESS, although *scipy_log_gaussian* only barely achieves the required 8 samples per chain with a bulk-ESS of 32. Those two likelihood functions, together with *scipy_per_month_mean_log_gaussian*, are the only three for which we can not calculate *ess_vehtari*. Their other ESS diagnostics are also comparatively bad with a max *esss_buithanh* of at least 1293, a min *ess_gelman* of at most 9, and a bulk-ESS of at most 170.

The next-most outstanding result is the very high ESS and consequently very low ESSS for *scipy_pre_mean_log_gaussian* with a bulk-ESS of 1101, a max *esss_buithanh* of 26, and at least 600 for the other ESS values. This is due to the large movements with a high acceptance rate that this likelihood function causes, which in turn makes the samples less correlated.

The remaining three likelihood functions result in fairly similar ESS values and thus form the middle ground between the extremes. For example, they have bulk-ESS values of 214 with *AR_log_gaussian*, 229 with *scipy_yearly_mean_log_gaussian*, and 235 with *scipy_post_mean_log_gaussian*. We show the last one's autocorrelation plot over multiple chains in Figure 6.7, where we can see that we reach 0 in all dimensions, the requirement to calculate *ess_vehtari* and *ess_gelman*, and fairly quickly in most dimensions as well.

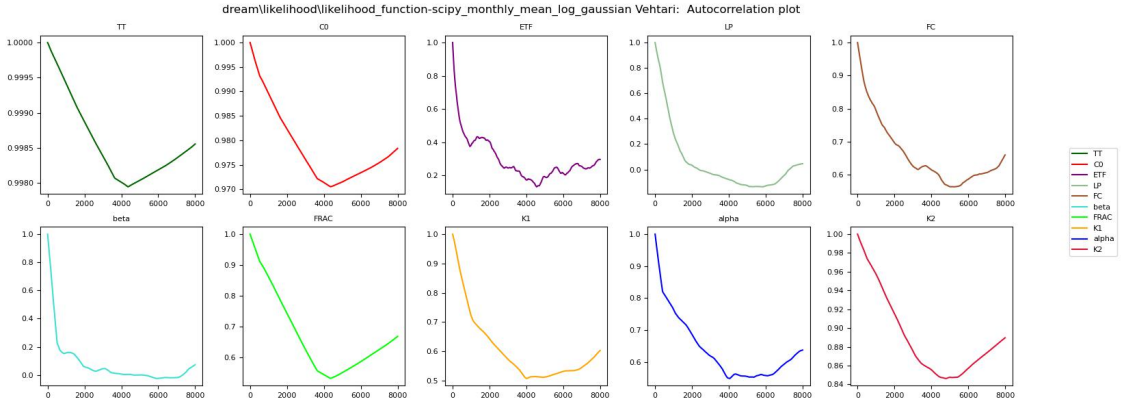


Figure 6.6: Bad multiple chain autocorrelation for *scipy_monthly_mean_log_gaussian*

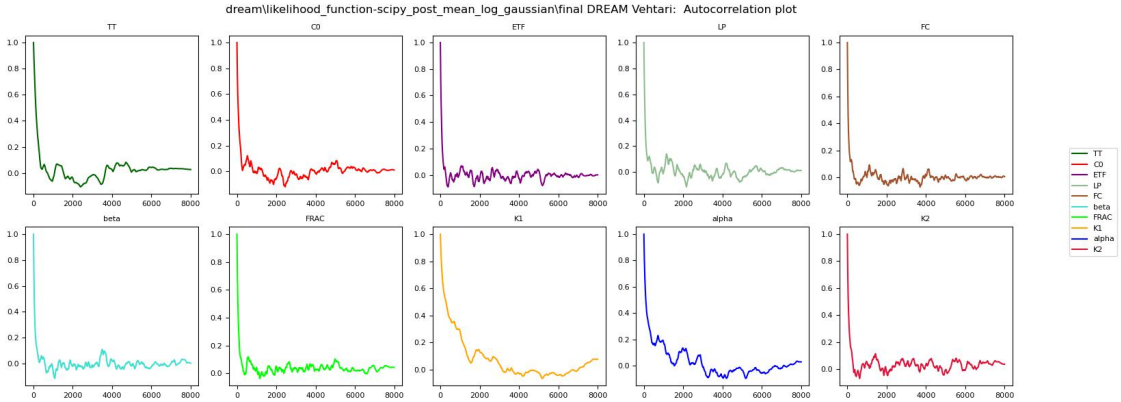


Figure 6.7: Average multiple chain autocorrelation using *scipy_post_mean_log_gaussian* function

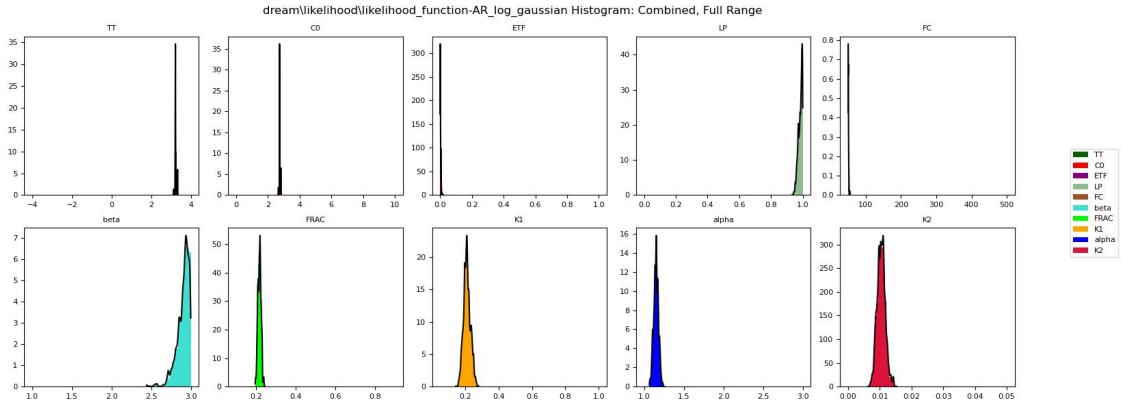


Figure 6.8: Histogram for a run using *AR_log_gaussian* likelihood function.

Therefore, even though the *scipy_post_mean_log_gaussian* function is not the best in one particular diagnostic, we continue to use it since it achieves good results in all diagnostics.

Additionally, we noticed that the run using the *AR_log_gaussian* likelihood function converges to the upper bound for *beta*, as can be seen in Figure 6.8. This might explain the "bad convergence" we mentioned with Figure 5.1 in the previous chapter, i.e., that it is not truly bad convergence, but just converges differently because of the other dimensions.

6.4 Likelihood Scale

We test eight different values for σ as the scale value in the *scipy_post_mean_log_gaussian* likelihood function: 20, 5, 2, 1, 0.5, 0.1, 0.05, and 0.01, of which we used 0.5 in the previous section.

Acceptance Rate

With this algorithm, we observe the expected behavior of a larger scale leading to a higher acceptance rate and vice versa: $\sigma = 20$ has the highest acceptance rate at 64.22%, followed by $\sigma = 5$ with 28.11%, down to $\sigma = 0.01$ with 2.00% and $\sigma = 0.05$ with 1.71%. The last two are the only ones that do not strictly follow this pattern, though only by a difference of 0.29%.

Error Functions

As in the previous section and chapters, the options with a high acceptance rate have bad error values, with $\sigma = 20$ having MAE values above 20 and RMSE around 40. Furthermore, the error values decrease as the scale decreases, except for the mode values, where $\sigma = 5$ achieves better results than $\sigma = 2$ as the sole exception. For MAE

values, using $\sigma = 1$ and lower results in mode MAE values of 10.45 or lower, and mean MAE below 10.44, so there is barely any difference between the mode and mean. Between the five best options, i.e., with a scale of 1, 0.5, 0.1, 0.05, and 0.01, there is only a difference of 0.3, so they can be treated as equal. It is similar for the RMSE values, where the same settings achieve the five best results with a mode RMSE of at most 16.93 and a mean RMSE of at most 16.92, and only a difference of 0.4 among the best five.

R-Convergence

The *rank/fold-*$\hat{R}$ values, on the contrary, worsen as the likelihood scale decreases, with $\sigma = 20$ having the best result of 1.003, followed by $\sigma = 5$ with 1.008, up to $\sigma = 0.1$ with 1.070. The lower likelihood scales of 0.05 and 0.01 have comparatively drastically higher *rank/fold-*$\hat{R}$ values at 1.60 and above, and thus have not converged according to the diagnostic.

ESS

Here we observe that the ESS increases as the likelihood scale increases, and thus the settings with a higher acceptance rate have higher ESS values again as in the previous section. We could not calculate an *ess_vehtari* value for $\sigma \in \{0.01, 0.05, 0.5\}$. These scale values also result in bulk-ESS and min and mean *ess_gelman* values below 70.

Similar to the acceptance rate, the result obtained with $\sigma = 20$ has the highest ESS results: a bulk-ESS of 1865, max *esss_buithanh* of 9, and at least 1000 for the other ESS values. Thus, the ESS values are even higher than for the result using the *scipy_pre_mean_log_gaussian* from the previous section.

Of the remaining four options, $\sigma = 0.5$ stands out negatively as its ESS values are roughly only half of the second worst among those four, with, e.g., a bulk-ESS of 195. The other options have a bulk-ESS of 408 for $\sigma = 1$, 413 for $\sigma = 2$, and 504 for $\sigma = 5$, and the max *esss_buithanh*, min *ess_vehtari*, and *ess_gelman* for $\sigma = 5$ are about twice as good as for $\sigma = 1$.

Thus, the small likelihood scales are unfit due to the R-convergence values and ESS values, and the large values because of the error functions. Thus, only the scale values of 1 and 0.5 deserve further consideration. Both the mean and mode error values for a likelihood scale of 1 are slightly worse by 0.2 for MAE and by 0.4 for RMSE than with a likelihood scale of 0.5, but better by 0.3 when using the test set to calculate the error values again. So, $\sigma = 1$ likely generalizes better than $\sigma = 0.5$. Furthermore, the former has a slightly better *rank/fold-*$\hat{R}$ value by 0.008 and ESS values by roughly a factor of 2. Therefore, we use $\sigma = 1$ for the rest of this chapter.

6.5 Starting Sample

We again use the same options for selecting starting samples as in Section 4.6.1, although we use the mean and mode values of samples we have obtained during this chapter so far, listed in Table 6.3. This time, the mode before and after burn-in is again the same, so we do not differentiate between them. Similarly, since the mean values vary only slightly between with and without burn-in, we use only the mean values after burn-in. We also see that for some dimensions, the mean and mode are even closer than for DRAM, which suggests better convergence. Furthermore, we observe actual movement for the *FC* dimension, as it is no longer just the mean value of its boundaries.

Table 6.3: The mean and mode for each dimension based on the previous samples after burn-in.

Name	Mean Burn-in	Mode
TT	1.78	1.91
C0	3.82	2.84
ETF	0.14	0.08
LP	0.95	0.99
FC	92.47	88.66
beta	1.20	1.00
FRAC	0.17	0.12
K1	0.22	0.05
alpha	1.72	1.88
K2	0.022	0.023

As expected, the different starting values have no real influence since the changes they cause are resolved during the burn-in phase, because the chains quickly converge. Thus, every starting sample results in almost the same MCSE values, and there is only a difference of 0.04 for mean MAE and 0.06 for mean RMSE between the best and worst for each error function.

For the mode values, there is a slightly larger difference of 0.47 for mode MAE between 10.72 for *random* and 10.25 with *prior mean*, and 0.43 for mode RMSE between 17.08 for *prior median* and 16.65 with *close default*. Interestingly, *random* has the second-best mode RMSE with 16.86, so the best and worst mode RMSE and mode MAE are almost reversed. Combined with the small difference between them anyway, this further shows that the difference is irrelevant.

The *rank/fold-*$\hat{R}$ values have even less difference with *spread* being the best at 1.006, followed by *close mean* with 1.007 up to *close default* as the worst with 1.017.

The ESS values differ not as much as in the previous sections, as well. For example, the smallest bulk-ESS is 300 with *random* followed by 366 with *spread*, whereas the largest is 469 with *prior mean*. The *random* method also achieves the worst results for mean

and max *ess_vehtari* and *ess_gelman* as well as max *esss_buithanh*. However, this setting results in average min *ess_vehtari* and *ess_gelman* values, which signifies lower variance among dimensions than other options have. The three *close* methods achieve the best max *esss_buithanh* values with 53 for *close default*, 54 for *close mean*, and 66 for *close mode*.

Examining the final ESSS, i.e., the worst result among all ESS diagnostics, most starting values result in an ESSS between 150 and 285, with *default* being the only one to exceed this with 483.

Considering all of this, we use the *close mean* method in the following sections since it achieves all-around good results (e.g., second-best rank fold, second-largest bulk-ESS, second-best mean MAE). But as we explained in the beginning, this decision is not too influential anyway, since there is overall barely any difference.

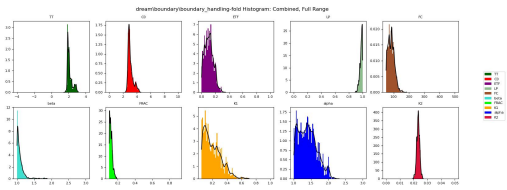
6.6 Boundary Handling

PyDREAM has its own method to ensure samples are within bounds, different from the methods we used so far. This method does the same as *eflect* once, and if a sample is still out of bounds afterwards, the sample is set to a random value within the boundaries for the dimensions in which it broke boundaries. We name this method *original* for simplicity, as it is the original method of the PyDREAM implementation. But we also modified the PyDREAM code slightly to include the *fold* and *bound* methods. We do not use the *eflect* method as the *original* method does essentially the same, since they only differ if samples are out of bounds after being reflected once. This happens for fewer than 20 samples per algorithm run (i.e., per 40,000 proposals) and is thus insignificant. Furthermore, we also do not test the *ignore* method as this requires more fundamental changes to the algorithm.

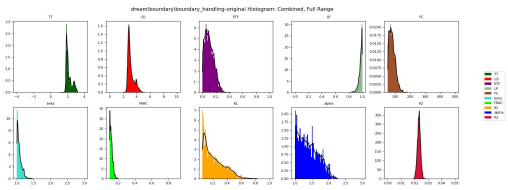
Generally, there is almost no difference between using *original* or *fold*, as we can see in their histograms in Figure 6.9. We also see that the histogram for *bound* is far more peaked at the boundary if the other methods converge towards the boundary. For example, it is apparent for *ETF*, but not noticeable for *K2*.

Acceptance Rate

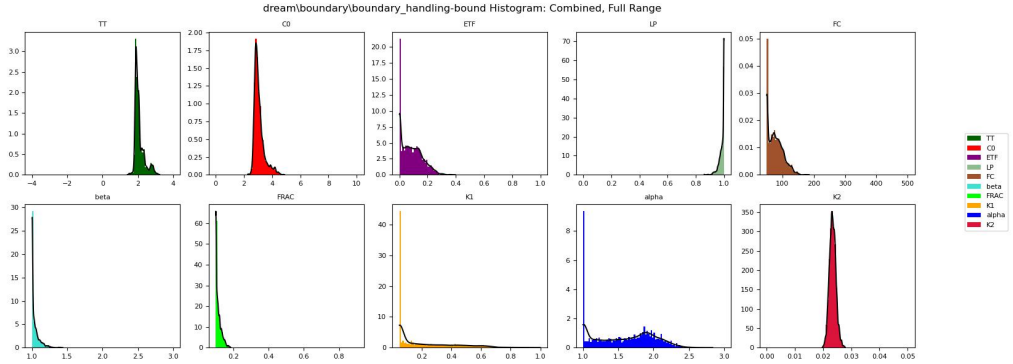
Unsurprisingly, *bound* has the highest acceptance rate with 20.03%, since if the current sample is at a boundary and the proposal is out of bounds, the proposal is mapped to the boundary. Thus, the proposal and last accepted sample are the same value and therefore have the same likelihood. Consequently, the proposal has an acceptance rate of 1, which guarantees acceptance. The next-highest is *original* with 15.96%, and finally *fold* with 8.68%, which is also expected since *fold* maps out-of-bound samples often to low probability areas, where they are rejected.



(a) Method *fold*



(b) Method *original*



(c) Method *bound*

Figure 6.9: Histograms of the runs with different methods to handle boundaries.

Error Functions

As discussed before, there is barely any difference between *original* and *fold*, with a maximum difference of 0.06, though *original* is slightly better overall. We observe some difference using *bound* as it is slightly better by 0.09 for mean MAE and mean RMSE, but worse by 0.9 for mode MAE and 2.1 for mode RMSE. This pattern persists, although slightly stronger, when using the test timespan. As the mode (i.e., the boundary value for some dimensions) is worse, but the mean is better than with *original* or *fold*, the high probability areas are likely very close to the boundaries for many dimensions, but not exactly at the boundary, as we already assumed before.

R-Convergence

Inspecting the *rank/fold-*$\hat{R}$ values, *fold* has the worst with 1.018, and *bound* has the best with 1.007 over *original* with 1.010. However, *bound* is possibly "cheating" again as the chains are more likely to converge to boundaries and not to the actual target distribution.

ESS

For the final ESS and ESSS values, i.e., by taking the worst results among all ESS diagnostics, *fold* has significantly worse results: *original* has a final ESSS of 152, followed by *bound* with 243, and lastly *fold* with 662. The latter was decided through min *ess_vehtari*, whereas the other two are based on min *ess_gelman*.

The *fold* method achieves roughly only half of the ESS that the other two methods achieve on average. But inspecting the relevant extreme values, it is even worse, as it falls behind by a factor of roughly 3 for min *ess_vehtari* and min *ess_gelman*, and 5 for max *esss_buithanh*. This is because *fold* has especially bad results for *beta*, with 48 for *ess_vehtari* and 51 for *ess_gelman*. Comparing the other two methods, *original* has a slightly larger ESSS for max *esss_buithanh* of 77 than *bound* with 70, which results in a smaller ESS by 42. But for the other extreme values, *original* performs significantly better with a min *ess_vehtari* of 218 compared to 137, and a min *ess_gelman* of 211 over 132.

Thus, while *original* and *bound* have almost identical mean *ess_vehtari*, mean *ess_gelman*, mean *split-*$\hat{R}$, and mean $\hat{R}$ values, *original* has a smaller difference between its minimum and maximum values for *ess_vehtari*, mean *ess_gelman*.

For this reason and the better performance of the error values, we keep using the *original* method to handle boundaries.

6.7 Probability for unity jumps

So far, we used the same γ_d as we disabled the possibility of $P(\gamma_d = 1) = 1$. Now, we test different probabilities from 0, 0.2, 0.5, and 0.8 to 1, where differences between other chains are not scaled when creating new proposals.

We observe that the acceptance rate decreases as $P(\gamma_d = 1)$ increases, from 16.25% down to 9.77%. This is to be expected, as a higher $P(\gamma_d = 1)$ leads to more large jumps, which are generally less likely to be accepted.

Error Functions and R-Convergence

We list the results for the error functions and *rank/fold-*$\hat{R}$ in Table 6.4. However, we do not discuss them any further since there are only minute differences, especially for the mean errors and *rank/fold-*$\hat{R}$ values, and we observe no pattern. Even when considering all diagnostics for R-convergence, there is less than a 0.057 difference between the best and the worst result.

Table 6.4: The error values and *rank/fold-*$\hat{R}$ values for the runs using different probabilities for unity jumps, sorted by the probability parameter.

$P(\gamma_d = 1)$	0	0.2	0.5	0.8	1.0	maximum difference
mean MAE	10.43	10.42	10.42	10.43	10.41	0.02
mode MAE	10.28	10.39	11.06	10.25	10.69	0.81
mean RMSE	16.89	16.88	16.88	16.87	16.87	0.02
mode RMSE	16.77	16.81	17.34	16.93	17.86	1.09
rank/fold-$\hat{R}$	1.013	1.013	1.004	1.017	1.012	0.012

ESS

Only the ESS values show real change between the different settings, as we show in Table 6.5. There, $P(\gamma_d = 1) = 0.2$ shows the best results for all diagnostics (including the min and max), except for min *ess_gelman*, where it is the second worst, although it has the best result for mean and max *ess_gelman*. A probability of 0.5 also has a good bulk-ESS value of 521, whereas a probability of 0 has the worst results in everything besides *esss_buithanh*, max *ess_vehtari*, and max *ess_gelman*, but the latter two are not relevant anyways.

Table 6.5: The relevant ESS and ESSS values.

$P(\gamma_d = 1)$	0	0.2	0.5	0.8	1.0	maximum difference
mean *esss_buithanh*	49	52	65	56	58	16
max *esss_buithanh*	65	61	90	60	61	30
mean *ess_vehtari*	246	335	277	269	289	88
min *ess_vehtari*	113	213	166	144	188	99
mean *ess_gelman*	219	314	283	270	281	94
min *ess_gelman*	77	105	194	155	189	117
bulk-ESS	270	543	521	327	357	272

Therefore, we use a probability of 0.2 for the tests in the next sections, which is the standard probability.

6.8 Crossover burn-in

So far, we have set the duration during which the crossover probabilities are adapted to 0. So, strictly speaking, we disabled adaptation. Now, we test the different burn-in lengths for crossover adaptation of 0%, 10%, 20%, and 50%. Therefore, we have no separate section in which we discuss the results between adaptation enabled and disabled, as this is essentially included here.

Furthermore, as the mean error values and R-convergence diagnostics have not changed significantly in the last four sections, we do not discuss them in the following sections unless they show more severe changes or interesting results.

Error Functions

Disabling the adaptation by using a length of 0% achieves the worst results for mode MAE with 10.70 and mode RMSE with 17.76. Lengths of 20% and 50% achieve the best results with mode MAE values of 10.29 and 10.41, and mode RMSE of 16.85 and 16.69, respectively.

ESS

For the ESS values, 10% has the worst results in all diagnostics besides mean *ess_vehtari* and bulk-ESS, where it is the second worst, and thus ahead of the results for using a crossover burn-in length of 20% by 9 and by 96, respectively. Using a length of 20% has the worst mean *ess_vehtari* with 296 and bulk-ESS with 368. However, this setting also results in the second-best results for max *esss_buithanh*, min *ess_gelman*, and min *ess_vehtari*, where it is at most 25 behind the best option. The overall best option according to ESS is to adapt during 50% of the entire execution, as it achieves the best results for all ESS diagnostics except mean *ess_vehtari* and mean *ess_gelman*. Doing no adaptation results in ESS being larger by 14 and 7 for these two diagnostics, respectively. The same pattern also mostly holds for bulk-ESS, besides a length of 20% having the worst value of 368 compared to 50% with 506.

Finally, by selecting the worst ESS results for each run, i.e., the sample size we use to perform the additional diagnostics evaluation, a length of 50% has the smallest (that is, the best) ESSS with 127, followed by 20% with 141, and 0% with 156. These worst results are all decided through min *ess_vehtari*. The run with a burn-in length of 10% has the worst final ESSS with a value of 265, which is decided through min *ess_gelman*. However, min *ess_vehtari* is only slightly better for the latter setting, and min *ess_gelman* is only marginally better for the first three options. Therefore, this result does not change if we only inspected min *ess_vehtari* or only min *ess_gelman*.

Thus, 20% has the best results for the error functions and second best for other relevant ESS diagnostics (max *esss_buithanh* and min *ess_vehtari* and min *ess_vehtari*) besides bulk-ESS, where it has the worst result by 138 to the best. At the same time, 50% achieves the best results for most ESS diagnostics, but only average results for the error functions. Therefore, we use a crossover burn-in length of 20%, i.e., 2000 iterations, as this also performs better for the test error values and fits with the normal burn-in length of 20% we use so far.

6.9 Number of Crossover Values

Next, we test three values for n_{CR}, namely 1, 3, and 5. The first setting means that CR contains only the value 1, so every dimension is updated in every proposal.

For the acceptance rates, we discover no pattern as $n_{CR} = 3$ has the highest rate with 14.58% and $n_{CR} = 1$ the lowest with 11.19%.

Similarly, there is no clear pattern for the error functions as $n_{CR} = 1$ has the best MAE results, $n_{CR} = 5$ has the best RMSE results, and $n_{CR} = 3$ has the median results for all but the mode RMSE. However, the largest difference between error values is only 0.39 for mode RMSE, and decreases to 0.02 for both mean MAE and mean RMSE, which is insignificant.

On the other hand, $n_{CR} = 3$ has the best ESS results except for mean, min, and max *esss_buithanh*, and bulk-ESS with 421. It is behind $n_{CR} = 5$ in these four diagnostics,

which has, for example, a bulk-ESS of 511. Lastly, $n_{CR} = 1$ has the worst ESS values, especially for the relevant extreme cases of max *esss_buithanh*, min *ess_vehtari*, and min *ess_gelman*, where it is worse by at least a factor of 2.

Since $n_{CR} = 3$ performs all around the best with consistently good results, we continue to use this value.

6.10 Snooker Probability

A snooker update is a special iteration step where the proposal is performed differently than with Equation 6.3: To make a proposal for the chain m with the current sample $z_{k,m}$, we gather the current state of three different chains a, b, and c. The sample from chain a indicates the direction in the parameter space in which the proposal is made, and thus forms a line through $z_{k,m}$. Then, we project the samples Z_b and Z_c orthogonally onto the line. The difference between those projected points defines how far the new proposal moves from $z_{k,m}$ along the line. [5, 17, 18]

So far, we have disabled this option by setting its probability to 0. Now, we test the probabilities of 0, 0.2, 0.5, 0.8, and 1.

Acceptance Rate

The acceptance rate decreases from 14.74% to 8.73% as the probability of snooker updates increases, suggesting that the snooker updates move too far. This is further reinforced by the fact that many more samples are still out of bounds after eflecting them once, compared to the previous sections, where it happened at most 20 times over 40,000 proposals.

Error Functions

There is no observable pattern between the snooker probability and the error values. For example, a probability of 0.5 has the worst mode RMSE with 17.82, and a probability of 0.8 has the best value with 16.65 for mode RMSE. However, the latter probability also has the worst result for mean RMSE and mean MAE.

R-Convergence

There is a comparatively large difference for the *rank/fold-*$\hat{R}$ values of 0.04 between the best probability of 0.2 and the worst of 0.8, though they are still well below a value of 1.2. The probabilities of 0 and 1.0 are close to the best and the worst, respectively.

ESS

The ESS values again show the most significant differences. Especially noticeable is that the probabilities of 0.5 and 0.8 are not even capable of calculating *ess_vehtari*. A

probability of 1.0 can calculate all diagnostics, but its results are still quite bad: This probability achieves a bulk-ESS of only 125, a min *ess_vehtari* of 59, a min *ess_gelman* of 63, and a max *esss_buithanh* of 364. Thus, the probability of 1.0 is the worst for *esss_buithanh* among all five options by more than a value of 100. The remaining probability settings of 0 and 0.2 achieve good results, but doing no snooker updates has consistently better results in every ESS diagnostic, such as a bulk-ESS of 525 compared to 0.2 with 380.

Hence, even though a probability of 0.2 has slightly better error values and *rank/fold-*$\hat{R}$ values, doing no snooker updates has significantly better ESS values. Thus, we continue to do no snooker updates. This surprisingly bad performance with snooker updates is likely because the snooker proposals differ more than necessary from the last accepted sample. This becomes apparent through the lower acceptance rate, and since many proposals are still out of bounds after being reflected once.

6.11 δ and Number of Chains

As mentioned before, we need at least $2\delta + 1$ chains for the DREAM algorithm. Therefore, we can not increase δ and keep using $M = 4$ chains. Thus, we try different δ and M values at the same time, as they are correlated. For the δ values of 1, 2, and 3, we require at least 3, 5, and 7 chains, respectively. We use 4, 6, and 8 chains to keep the number of chains even as we have so far.

Acceptance Rate and Execution Time

We observe a decrease in acceptance rate as δ and M increase from 15.3% to 12.6% to 12.0%. However, since more chains result in more samples, the overall number of unique samples increases from 4906 to 7708.

The execution time also increases with M: from 8994s ($\approx$ 2h30min) with 4 chains to 10832s ($\approx$ 3h) with 6 to 13180s ($\approx$ 3h40min) with 8.

Error Functions

We obtain the best results for mode error functions with $\delta = 2$ with 10.31 and 16.65 for MAE and RMSE. The next best is $\delta = 3$ with 10.47 and 16.69, followed by $\delta = 1$ with 10.72 and 17.72. So, while there is a better choice regarding the error functions, the difference in results is negligible, especially due to the inherent randomness of MCMC.

ESS

The multi-chain diagnostics show an increase in ESS as δ and M increase, e.g., for bulk-ESS from 453 to 617 to 751. However, if we normalize these values to ESS per chain, then we obtain 113, 102, and 56. The same pattern persists for the other multi-chain

diagnostics where $M = 8$ has the highest overall ESS, but the smallest ESS per chain, and vice versa for $M = 4$, which has almost twice the ESS per chain as the former.

Inspecting the *esss_buithanh* values that are by definition not dependent on the number of chains, we see the same behavior with $M = 4$ and $\delta = 1$ performing the best with 69, and $M = 8$ and $\delta = 3$ the worst with 114 for max *esss_buithanh*. Thus, we can also assume that the increase in δ alone worsens the ESS values.

We only observe a small improvement for the error functions, but every other diagnostic worsens with increasing M and δ. Therefore, we continue with $M = 4$ chains and $\delta = 1$.

6.12 Burn-in Length

We use the same burn-in lengths of 0%, 20%, 50%, and 80% as in the previous two chapters.

Error Functions

Using all samples has the worst mode results with 12.22 and 20.17, whereas 20% has the best with 10.72 and 17.72 for MAE and RMSE. Removing the first 80% has the second-worst mode MAE with 11.15 after 50% with 10.95. However, for the mode RMSE, they swap positions as 50% has the second worst mode RMSE with 18.40, and 80% the second best with 18.30. Since each option results in a different value, the mode of the samples also changes frequently, which shows that no chain gets stuck.

A similar behavior also occurs when using the test dataset, where 0% is the worst, followed by 80%. The last two options are very close, with a difference of at most 0.04, where 20% is the best for the test mode MAE, and 50% the best for test mode RMSE.

ESS

A burn-in length of 20% achieves the best results in all ESS diagnostics except min *esss_buithanh*, which is not as important, with a difference of at least 100 for most multi-chain diagnostics, and a slight advantage for *esss_buithanh*. However, if we calculate the ESSS for the multi-chain diagnostics, i.e., normalize the ESS values with the number of samples, 20%, 50%, and 80% achieve almost the same values. This result is especially noticeable for bulk-ESS, mean *ess_vehtari*, and mean *ess_gelman*, where the three settings differ by at most 3. Furthermore, no burn-in results in the largest ESSS values by almost a factor of 2 for most diagnostics. Especially the *esss_gelman* values are almost three times worse than for the other burn-in lengths.

Thus, we continue to use a burn-in length of 20% as it generally results in the best error values, although only slightly, close to the best ESSS and the highest ESS. Additionally, it has the best *rank/fold-*$\hat{R}$ value, though they are all well below 1.2.

6.13 Summary

The best parameters for the DREAM algorithm according to our tests are summarized again in Table 6.6. The results we obtained with those parameters are listed in Table 6.7 as well as in Figure 6.10. In Figure 6.10a, we can nicely see that the samples converge to a small area in most dimensions, but still move around within that area. This causes the histogram in Figure 6.10b, which is peaked but not too steep and thin. In Figure 6.10d, we can see that the first plotted value is already around 1.175 and thus below 1.2, so it can already be treated as having converged. Furthermore, this plot does not move above this value, only rising once at around iteration 3000 (1000 in the plot because we discard the first 2000 iterations) and then stays relatively constant from iteration 5000 onwards. Comparing the plots in Figures 6.10e and 6.10f, we can see that the autocorrelation values for single chain and multiple chains both quickly move towards and sway around 0, with the latter having smaller variations. Therefore, fewer iterations would likely suffice for the DREAM algorithm.

Table 6.6: The final parameters of the DREAM algorithm.

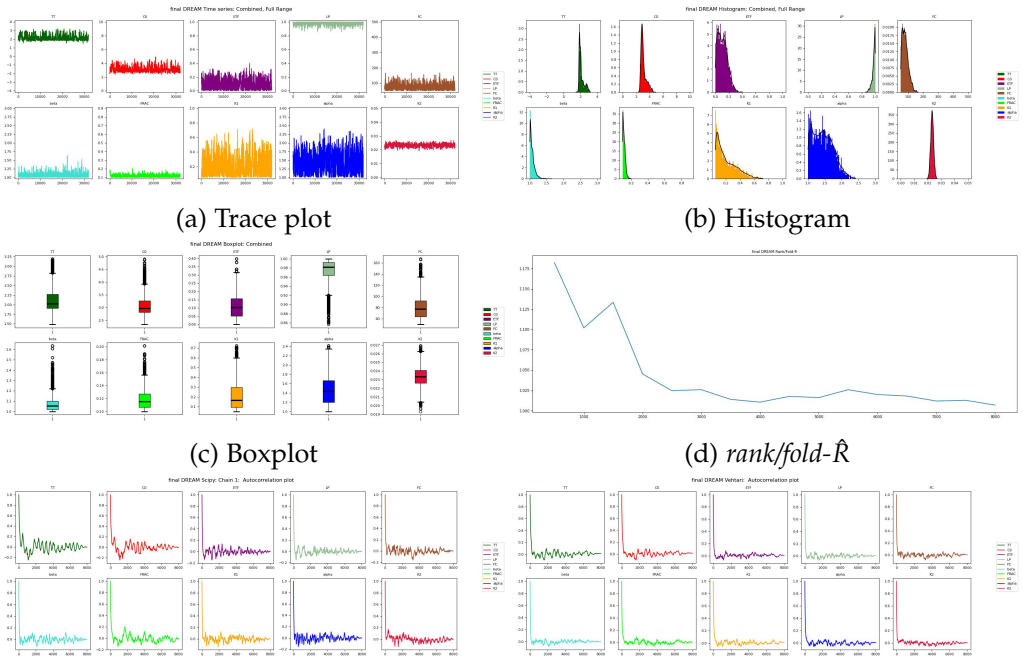
Parameter Name	Value
Likelihood Function	*scipy_post_mean_log_gaussian*
Likelihood Scale	1
Starting Sample	close mean
Boundary Handling	Original
$P(\gamma_d = 1)$	20%
adapt_crossover	True
crossover_burn-in	20%
n_{CR}	3
δ	1
snooker probability	0%
Number of Chains	4
Burn-in Length	20%

To summarize, only the choice of the likelihood function and likelihood scale has a large impact on the error functions and R-convergence values. Only the mode error values change slightly with other parameters, although this might be because of the randomness of MCMC and volatility of the mode, and not the influence of the different parameter values. Upon setting those two options to good, fitting parameters, the mean error values and R-convergence values are consistently good, especially for the latter compared to MH and DRAM.

However, the ESS and ESSS values do change significantly for almost all options. We also notice that bulk-ESS gives a higher ESS than mean *ess_vehtari* or mean *ess_gelman*, presumably because by definition it averages over the dimensions and thus compensates outliers. The last two diagnostics often result in similar ESS values. Similarly, the

Table 6.7: The final results using the mean and mode of samples obtained through the settings in Table 6.6.

Name	Mean	Mode
TT	2.14	1.85
C0	3.11	2.81
ETF	0.11	0.14
LP	0.97	0.98
FC	80.36	50.44
beta	1.07	1.14
FRAC	0.12	0.11
K1	0.21	0.06
alpha	1.45	1.12
K2	0.023	0.023
MAE	10.41	10.72
RMSE	16.86	17.72
mean *esss_buithanh*	53	
bulk-ESS	453	
mean *ess_vehtari*	331	
mean *ess_gelman*	329	
rank/fold-$\hat{R}$	1.007	
mean *split-*$\hat{R}$	1.014	



(a) Trace plot (b) Histogram

(c) Boxplot (d) *rank/fold-*$\hat{R}$

(e) Single chain autocorrelation (f) Multiple chains autocorrelation

Figure 6.10: Diagrams for the final DREAM run.

difference between bulk-ESS and the ESS based on max *esss_buithanh* is relatively small. This also suggests that our choice of 0.5 as the threshold for *esss_buithanh* is not too high. Thus, *ess_vehtari* and *ess_gelman* are the deciding factor of the ESS for most of the additional diagnostic evaluations, since they have the worst and thus strictest ESSS results.

Furthermore, unlike most of the results obtained through MH and DRAM, using the mean of the samples results in better error values than using the mode. However, a good mean result is more valuable than a good mode result since the former represents overall good performance, whereas the latter can just be an outlier. Additionally, since the DREAM algorithm has a fairly high acceptance rate and because the mode of samples is the sample for which the most new proposals are rejected, the mode for DREAM only has a few occurrences and thus is not indicative of the entire run. This is not the case for MH and DRAM due to their lower acceptance rates.

We also see here that the DREAM results are all around better than the results from MH or DRAM, but the former have a higher MCSE. This shows the aforementioned difficulty in evaluating based on MCSE values, which is why we have not paid much attention to them in these chapters.

7 HBV-SASK Model Evaluation

Now that we have decided on good settings for our three MCMC algorithms, we train them on different training datasets and compare the results for different evaluation datasets.

7.1 Datasets

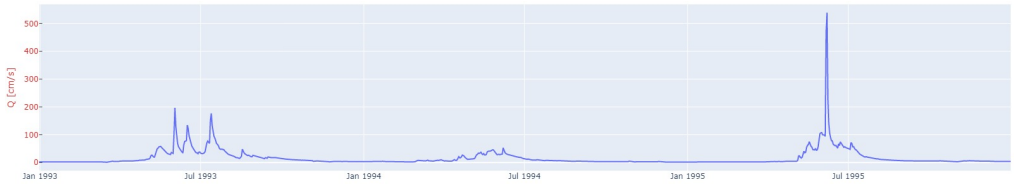
Since the Banff basin has such consistent behavior, we only use one training set and one evaluation set for it. For training, we use the timespan we have already used in the previous settings and thus reuse the samples obtained in the last runs for each algorithm. For the evaluation set, we select the last 20 recorded years and use a spin-up length of 3 years, as with the training set.

Additionally, we use multiple but smaller training and evaluation datasets for the Oldman basin, since its behavior is less consistent but has fewer recorded years. We select one inconsistent timespan with few high peaks and one relatively calm and consistent timespan each for training and evaluation, with one year spin-up and three more years for actual evaluation:

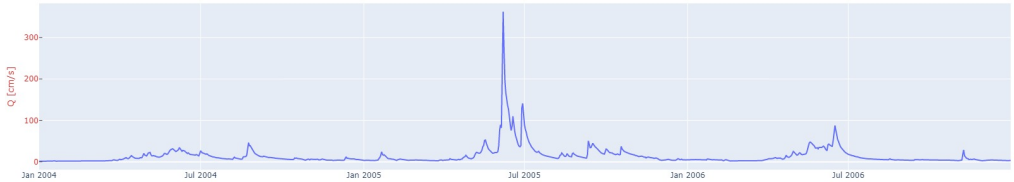
- The peaked training set ranges from 1992 to 1995, shown in Figure 7.1a
- The peaked evaluation set ranges from 2003 to 2006, shown in Figure 7.1b
- The calm training set ranges from 1982 to 1985, shown in Figure 7.1c
- The calm evaluation set ranges from 1987 to 1990, shown in Figure 7.1d

7.2 Parameters

The mean and mode of samples for the three training sets as well as the corresponding error values are listed in Table 7.1 for the MH algorithm, in Table 7.2 for the DRAM algorithm, and in Table 7.3 for the DREAM algorithm. There, we see that the Banff training set is the most difficult as it causes the highest error values, except for the mean RMSE on the peaked Oldman dataset with the MH algorithm. We also see that the calm Oldman dataset is the easiest and simplest to simulate, which is not surprising. Additionally, we observe a larger difference between MAE and RMSE values for the peaked Oldman dataset than for the other datasets. This difference suggests that there are a few instances with a large difference between the simulated and observed data,

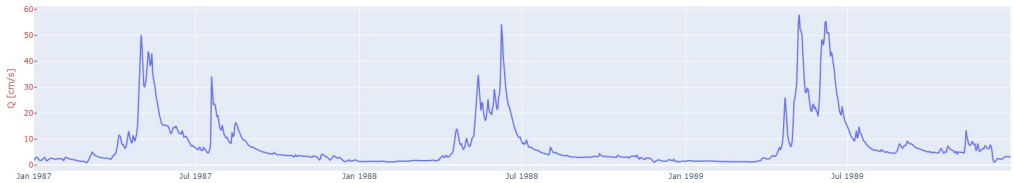


(a) Peaked training set



(b) Peaked evaluation set

(c) Calm training set



(d) Calm evaluation set

Figure 7.1: The four timespans selected for the Oldman basin.

suggesting that the few peaks are not approximated correctly. Furthermore, the mode values for the MH and DRAM algorithms result in lower error values than the mean values, whereas it is the other way around for DREAM.

Table 7.1: The mean and mode values obtained on the three training sets with the MH algorithm and the corresponding error values.

	Banff		Peaked Oldman		Calm Oldman	
	Mean	Mode	Mean	Mode	Mean	Mode
TT	2.46	1.50	2.51	3.21	1.31	1.34
C0	3.83	2.41	1.92	1.99	2.38	1.23
ETF	0.10	0.17	0.08	0.01	0.29	0.43
LP	0.99	1.00	0.59	0.97	0.38	0.61
FC	277.11	152.17	277.58	384.18	272.57	308.83
beta	1.37	1.04	1.82	2.10	1.87	1.97
FRAC	0.11	0.10	0.42	0.32	0.19	0.10
K1	0.26	0.22	0.47	0.96	0.30	0.99
alpha	2.06	1.54	2.05	2.87	1.69	1.91
K2	0.028	0.027	0.043	0.048	0.025	0.036
MAE	11.85	10.80	7.39	7.25	3.10	2.11
RMSE	18.53	17.21	19.00	16.39	5.62	3.06

Unsurprisingly, when running the evaluation datasets with parameters from all training sets and comparing their MAE and RMSE, the parameters trained on the same basin perform better. This is further enhanced if the training set has a similar behavior as the evaluation set (i.e., both calm or both peaked). For example, the parameters trained on the Banff basin perform worse on both Oldman evaluation sets than parameters trained on either of the Oldman training sets. However, the parameters trained on the Banff basin used on the Oldman basin still have smaller error values than the same parameters evaluated on the Banff basin, which further shows that the Banff basin is, in general, more difficult to predict. Unexpectedly, those parameters result in similar RMSE values for the Oldman basin to those trained on the Oldman basin, although the latter have lower MAE values. These results suggest that the former parameters make a few small errors but cause no large divergences from the measured data.

The parameters trained on the calm Oldman dataset perform better on the Banff dataset than the parameters trained on the peaked Oldman dataset. We assume that this is because the Banff basin's streamflow has a more consistent, periodic behavior, which is more similar to the calm dataset than the peaked dataset. Furthermore, the parameters obtained through the DREAM algorithm on one basin surprisingly have worse error values when evaluated on the other basin than with MH and DRAM. This suggests that DREAM generalizes worse, and thus essentially overfits, presumably due to its fast convergence. Interestingly, the parameters trained on the calm Oldman dataset also perform better for the peaked Oldman evaluation set than those trained on the

Table 7.2: The mean and mode values obtained on the three training sets with the DRAM algorithm and the corresponding error values.

	Banff		Peaked Oldman		Calm Oldman	
	Mean	Mode	Mean	Mode	Mean	Mode
TT	2.37	2.77	2.89	2.53	1.45	2.10
C0	3.61	4.39	1.72	1.35	2.00	1.55
ETF	0.14	0.19	0.02	0.04	0.43	0.15
LP	0.99	0.99	0.60	0.69	0.40	0.55
FC	267.41	268.70	259.65	277.43	251.97	296.89
beta	1.42	1.03	1.74	2.10	1.49	1.63
FRAC	0.11	0.10	0.49	0.65	0.29	0.68
K1	0.21	0.16	0.18	0.08	0.19	0.05
alpha	1.74	1.30	1.84	1.49	1.63	1.01
K2	0.027	0.028	0.043	0.037	0.020	0.001
MAE	11.57	10.30	7.05	5.73	3.02	1.45
RMSE	18.07	17.45	17.18	13.85	5.93	2.62

Table 7.3: The mean and mode values obtained on the three training sets with the DREAM algorithm and the corresponding error values.

	Banff		Peaked Oldman		Calm Oldman	
	Mean	Mode	Mean	Mode	Mean	Mode
TT	2.14	1.85	2.44	2.50	1.69	1.63
C0	3.11	2.81	1.27	1.27	1.48	0.38
ETF	0.11	0.14	0.19	0.32	0.47	0.06
LP	0.97	0.98	0.25	0.07	0.56	0.64
FC	80.36	50.44	130.17	132.03	385.61	437.42
beta	1.07	1.14	1.92	2.56	1.71	2.36
FRAC	0.12	0.11	0.60	0.55	0.22	0.67
K1	0.21	0.06	0.07	0.06	0.43	0.10
alpha	1.45	1.12	1.55	1.67	1.80	1.04
K2	0.023	0.023	0.030	0.026	0.035	0.005
MAE	10.41	10.72	5.57	5.68	2.29	3.60
RMSE	16.86	17.72	13.45	13.74	3.68	6.63

peaked Oldman dataset. This is presumably because the majority of the peaked dataset is still calm with low streamflow values.

7.3 Plots

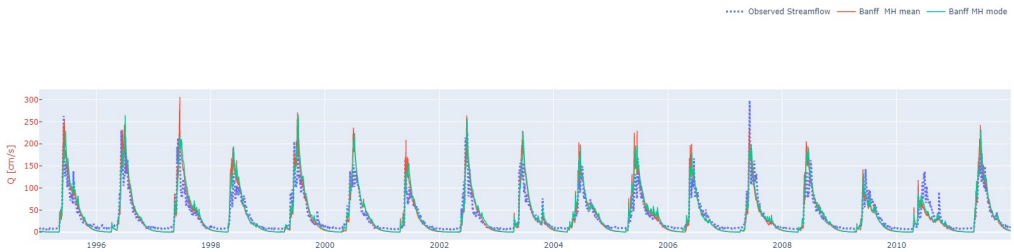
For each evaluation time span, we create three plots. Each plot uses the parameters trained with either the MH, DRAM, or DREAM algorithm on the corresponding training time span to execute the HBV-SASK model.

The plots for the Banff evaluation set are shown in Figure 7.2. There, we see that the simulations often overshoot during the peaks, which is especially noticeable for MH. For MH and DREAM, the mean parameters generally have higher peaks, whereas DRAM behaves mostly similarly between the mean and mode parameters. But overall, all parameters perform similarly well and mostly follow the measured values correctly by rising and falling at the accurate time. The only exception to this is in 2010, where the simulated values all rise slightly earlier as well as less than the measured values.

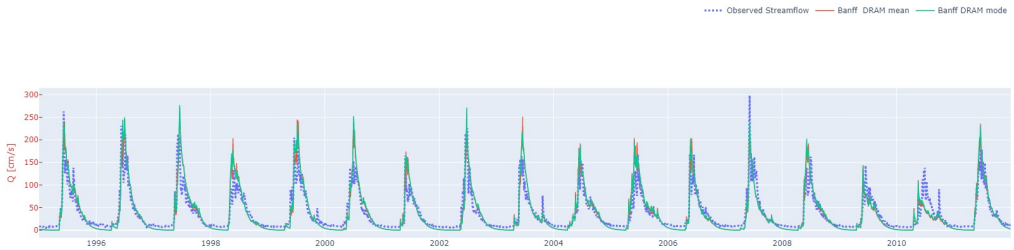
Figure 7.3 shows model runs on the peaked Oldman evaluation set. Although the other two algorithms also have similar behaviors, DREAM has the least differences for this dataset between its simulations when using the mean and mode parameters. The only noticeable difference is that the mode parameters do not capture the last peak. Here, especially for the mode values, MH stands out again by diverging from the measured values and other simulated values between the peaks, for example, during July 2005, after the highest peak. The error values also reflect this, as MH has the largest error values, whereas DREAM has slightly lower errors than DRAM. However, they all approximate the highest peak quite closely.

Lastly, the plots for the calm Oldman evaluation set are in Figure 7.4. Here, MH has arguably the least difference between mean and mode, though they are not very similar either. However, DREAM has a noticeable difference as the mode performs very badly by rising too late for every peak. We also see that no parameter set approximates the peak in the middle around June 1988 correctly, whereas the parameters all fit the calm areas very well. Furthermore, the mean parameters obtained with DRAM are the only parameters that cause the simulations to significantly overshoot during a peak, specifically for the first peak.

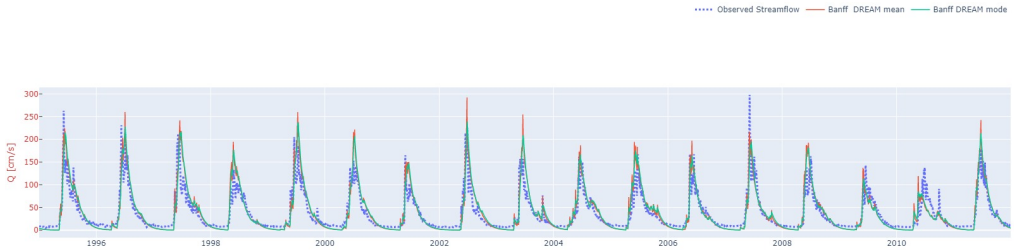
In addition to the previous nine plots, we also plot the results on the entire datasets for each basin with three years of spin-up and with the mean and mode parameters that achieve the lowest error per training set. These are shown in Figures 7.5 and 7.6. We can see that the parameters trained on the peaked Oldman dataset often surpass the observed peak of the Banff basin, which is not surprising, as they are trained for sudden high peaks. For the Oldman basin, we can not perceive that the parameters trained with the Banff data perform significantly worse than those trained on the Oldman basin, which is expected due to the relatively close error values we mentioned before.



(a) MH

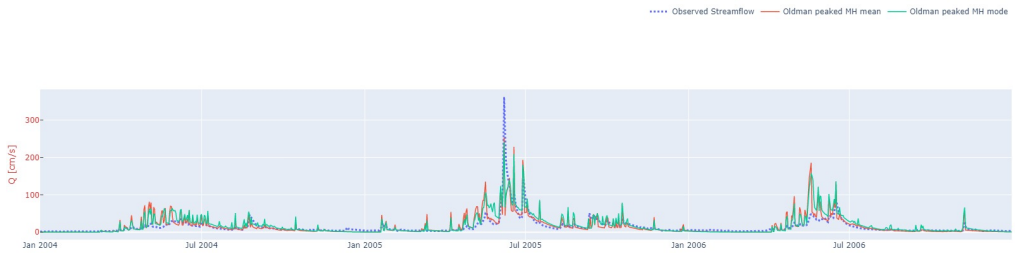


(b) DRAM

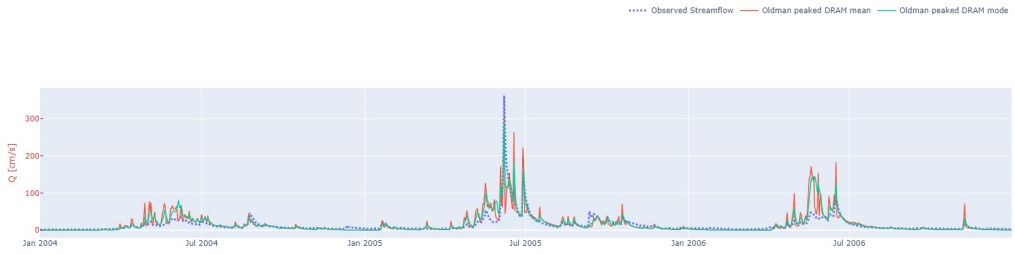


(c) DREAM

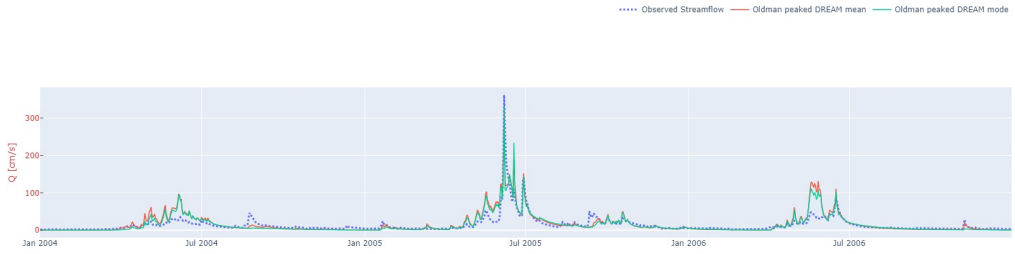
Figure 7.2: The parameters obtained with the Banff training dataset run on the Banff evaluation set.



(a) MH

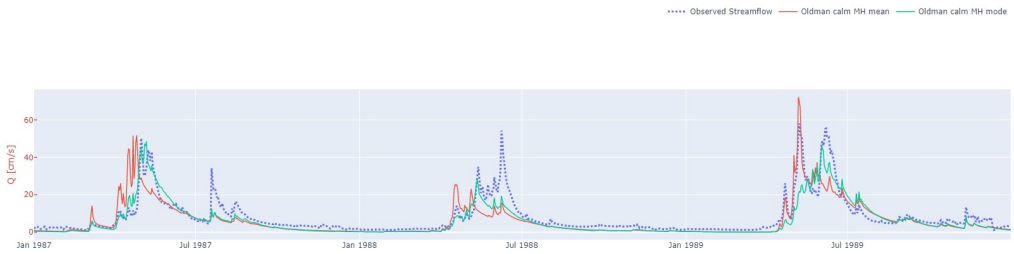


(b) DRAM

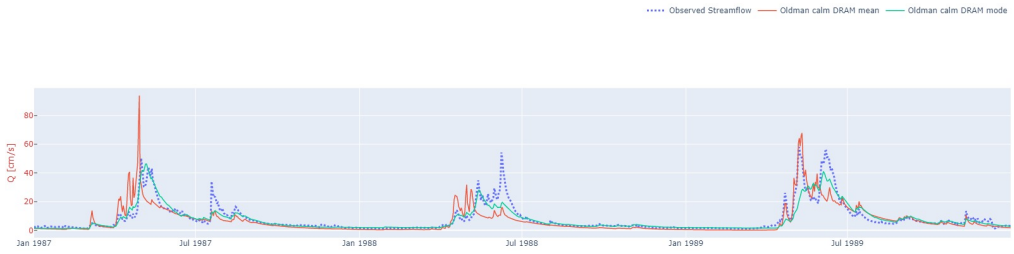


(c) DREAM

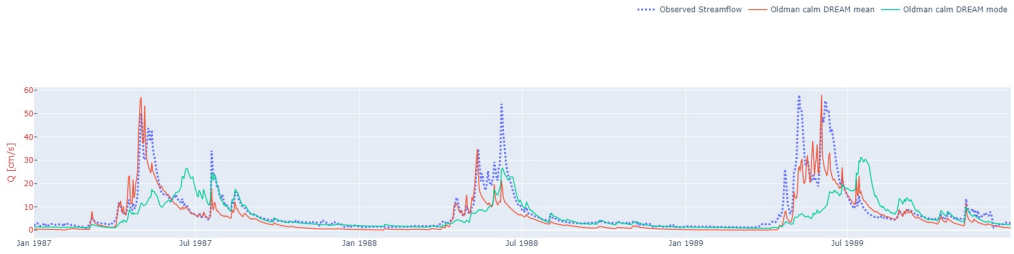
Figure 7.3: The parameters obtained with the peaked Oldman training dataset run on the peaked Oldman evaluation set.



(a) MH

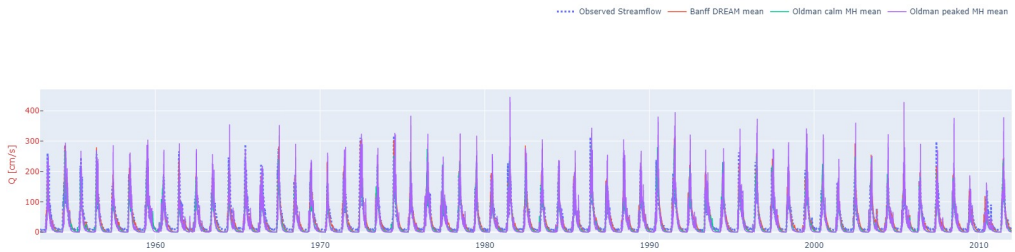


(b) DRAM

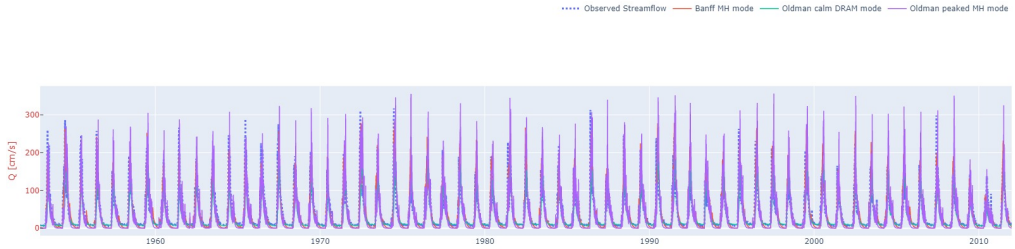


(c) DREAM

Figure 7.4: The parameters obtained with the calm Oldman training dataset run on the calm Oldman evaluation set.

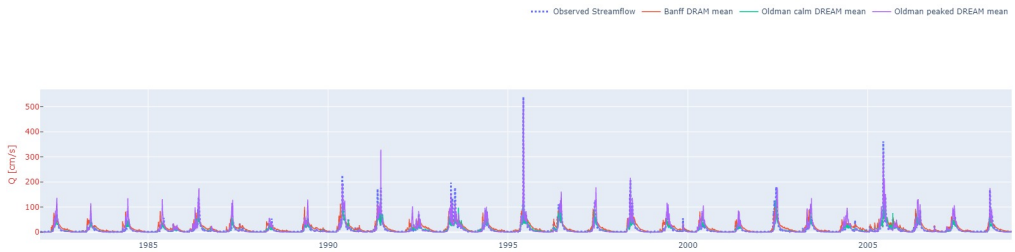


(a) Mean

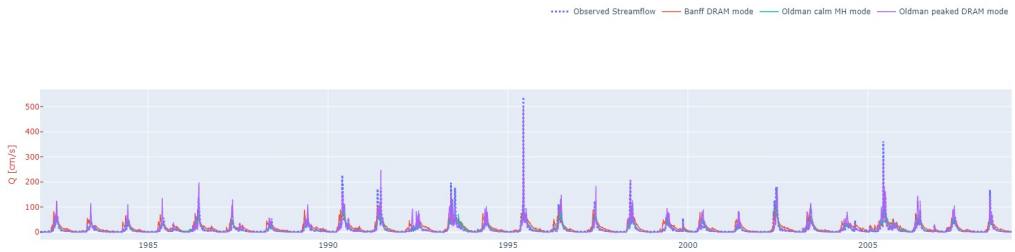


(b) Mode

Figure 7.5: The best parameters obtained for each training dataset run on the entire Banff dataset.



(a) Mean



(b) Mode

Figure 7.6: The best parameters obtained for each training dataset run on the entire Oldman dataset.

8 Conclusion

We have discussed the basics of three MCMC algorithms – the MH, DRAM, and DREAM algorithms.

The MH algorithm is the basic MCMC algorithm on which the other algorithms are built. It only creates a proposal, calculates its acceptance rate, and then either accepts or rejects the proposal in each iteration. The proposal is created by a proposal distribution, which is a multivariate Gaussian with a fixed covariance matrix in our case.

The DRAM algorithm does mostly the same as the MH algorithm, but it adapts its proposal covariance matrix as the algorithm progresses, and makes multiple proposals in one iteration if the first proposal is not accepted. Consequently, it has a longer execution time, but the choice of the proposal covariance matrix is not as important as for the MH algorithm.

Lastly, the DREAM algorithm is the only algorithm that naturally works with and requires multiple chains. It does not use a proposal distribution but instead makes new proposals based on the current states of other chains. Furthermore, this algorithm contains multiple additional improvements, such as updating only a subset of the dimensions with a new proposal.

We tested multiple parameter settings for each algorithm with the HBV-SASK model on the Banff basin as a training dataset. We evaluated those results based on MAE and RMSE error functions, R-convergence, and ESS diagnostics, as well as histograms, trace plots, and boxplots combined with the acceptance rate and execution time. We observed that the choice of the likelihood function and the proposal covariance matrix has the most impact on the MH and DRAM algorithms. The likelihood function and likelihood scale are the most influential for the DREAM algorithm, which uses no proposal distribution. Furthermore, all algorithms achieved similar error values with their best settings. The MH algorithm performed worse on the ESS and R-convergence diagnostics, where the DREAM algorithm performed exceptionally well in comparison. We also discovered that the ESS and especially R-convergence diagnostics can be tricked fairly easily, i.e., achieve good results for bad samples. This can occur, for example, when most samples are accepted or the chains converge to a fixed value different from the target distribution, such as a boundary. However, the error functions and diagrams could expose such behavior.

After obtaining fitting parameter values for each algorithm according to these metrics, we used those settings for each MCMC algorithm to train the model's parameters on additional training sets for the Oldman basin. Then, we used those parameters for the HBV-SASK model on further evaluation sets of both the Banff and Oldman basins. We discovered that the other two algorithms outperform the DREAM algorithm in the

error functions when evaluated on data with different behavior from the training data. However, if trained on similar data, the mean of the samples obtained through DREAM as model parameters achieves better results than the mean of the other two algorithms.

As follow-up work, one could put more focus on testing different likelihood functions, likelihood scales, and proposal covariance matrices, while keeping the other parameters fixed. For example, one could use custom proposal covariance matrices with a larger variance for the *FC* dimension. One could also use multiple or different training datasets than we used while testing the MCMC parameters, or use the same settings multiple times, as we sporadically did, to make the results more reliable and well-founded. Other MCMC algorithms with different ideas from those we used can be tested as well. Another idea is to create a fixed ranking system that includes all diagnostics to give consistent evaluations. With such a system, results could also be compared objectively and automatically, which would in turn allow for more test cases, as the results would not have to be compared manually.

List of Figures

List of Tables

Bibliography

[1] H. V. Gupta and S. Razavi. "Revisiting the basis of sensitivity analysis for dynamical earth system models." In: *Water Resources Research* 54.11 (2018), pp. 8692–8717.

[2] C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag, 2006. ISBN: 0387310738.

[3] T. Bui-Thanh. "A gentle tutorial on statistical inversion using the Bayesian paradigm." In: *ICES Report* (2012), pp. 12–18.

[4] H. Haario, M. Laine, A. Mira, and E. Saksman. "DRAM: efficient adaptive MCMC." In: *Statistics and computing* 16 (2006), pp. 339–354.

[5] J. A. Vrugt. "Markov chain Monte Carlo simulation using the DREAM software package: Theory, concepts, and MATLAB implementation." In: *Environmental Modelling & Software* 75 (2016), pp. 273–316.

[6] S. Razavi, R. Sheikholeslami, H. V. Gupta, and A. Haghnegahdar. "VARS-TOOL: A toolbox for comprehensive, efficient, and robust sensitivity and uncertainty analysis." In: *Environmental modelling & software* 112 (2019), pp. 95–107.

[7] C. Zhou. "Efficient Bayesian Inference of Hydrological Model Parameters: Implementation of a Parallel Markov Chain Monte Carlo Approach." MA thesis. Technische Universität München, 2024.

[8] J. Kaipio and E. Somersalo. *Statistical and computational inverse problems*. Vol. 160. Springer Science & Business Media, 2006.

[9] A. Vehtari, A. Gelman, D. Simpson, B. Carpenter, and P.-C. Bürkner. "Rank-normalization, folding, and localization: An improved $\hat{R}$ for assessing convergence of MCMC (with discussion)." In: *Bayesian analysis* 16.2 (2021), pp. 667–718.

[10] F. S. Al-Duais and R. S. Al-Sharpi. "A unique Markov chain Monte Carlo method for forecasting wind power utilizing time series model." In: *Alexandria Engineering Journal* 74 (2023), pp. 51–63.

[11] C. J. Geyer. "Introduction to Markov Chain Monte Carlo." Slides from `https://www.stat.umn.edu/geyer/mcmc/`. 2003.

[12] C. J. Geyer. "Introduction to markov chain monte carlo." In: *Handbook of markov chain monte carlo* 20116022.45 (2011), p. 22.

[13] A. Gelman and D. B. Rubin. "Inference from iterative simulation using multiple sequences." In: *Statistical science* 7.4 (1992), pp. 457–472.

[14] A. Gelman, J. B. Carlin, H. S. Stern, D. B. Dunson, A. Vehtari, and D. B. Rubin. *Bayesian data analysis Third edition*. Chapman and Hall/CRC, 1995.

[15] C. J. Geyer. "Practical markov chain monte carlo." In: *Statistical science* (1992), pp. 473–483.

[16] R. Pederson. *Maximum Likelihood Estimation of the AR(1) Model*. 2017.

[17] E. Laloy and J. A. Vrugt. "High-dimensional posterior exploration of hydrologic models using multiple-try DREAM (ZS) and high-performance computing." In: *Water Resources Research* 48.1 (2012).

[18] C. J. Ter Braak and J. A. Vrugt. "Differential evolution Markov chain with snooker updater and fewer chains." In: *Statistics and Computing* 18 (2008), pp. 435–446.