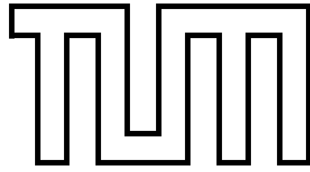


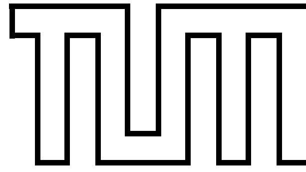
SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Efficient Bayesian Inference of Hydrological Model Parameters: Implementation of a Parallel Markov Chain Monte Carlo Approach

Chengjie Zhou



SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Efficient Bayesian Inference of Hydrological Model Parameters: Implementation of a Parallel Markov Chain Monte Carlo Approach

Effiziente Bayesschen Inferenz der Parameter von der hydrologischen Modelle: Implementierung eines Markov-Chain-Monte-Carlo-Verfahrens im parallelen Modus

Author:	Chengjie Zhou
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor:	Ivana Jovanovic Buha, M.Sc. (hons)
Date:	09.08.2024

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

Munich, 09.08.2024 Chengjie Zhou

Acknowledgements

I would like to take this opportunity to thank M.Sc. Ivana Jovanovic Buha for her guidance. Her explanations and advice greatly helped me understand the topic of Markov chain Monte Carlo. Our regular meetings and communications motivate me to keep researching this interesting topic. I would also like to thank professor Univ.-Prof. Dr. Hans-Joachim Bungartz for introducing me to numerical computing and scientific computation via lectures and offering me this amazing opportunity to write this thesis in his chair. Many thanks for M.Sc. Sebastian Wolf for giving me insights into the general parallel Metropolis-Hastings algorithm.

I am grateful for my informatics teacher from my high school, Andreas Reisenbauer, from BRG9 Vienna. He exposed me to the exciting field of informatics and allowed me to feel the joy of learning new things, both inside and outside of the field of informatics. His support inspires me to keep working for continuous growth and discovery in my academic career and personal pursuits.

I also have great appreciation for all my friends at the university. With our open communication and their warm-hearted support, I feel extra motivated to work on my thesis. Their constant encouragement and willingness to support me have made these challenging three years of bachelor's enjoyable and rewarding for me.

Last but not least, I would like to thank my whole family and my partner for their continuous support. During the entire process of writing this thesis, I came across a few difficulties in my thesis and life. Their advice helps me a lot to cope with these difficult situations.

Abstract

Bayesian inference of hydrological model parameters is crucial for improving the accuracy and reliability of hydrological model executions. This thesis presents the implementation of the Markov Chain Monte Carlo (MCMC) approach to enhance the accuracy and the efficiency of Bayesian parameter estimation, with a predominant focus on the parallel version of the algorithms. Results regarding the accuracy and efficiency of the Bayesian inference are analyzed through comparison metrics and displayed using detailed visualizations so that the relationship between algorithm implementation variants and the results can be comprehended. Besides, the relationship between the training time series for the Markov chain Monte Carlo algorithms is also considered. By investigating these aspects of the algorithms and the data set, more insights regarding the performance and the result of Bayesian inference can be gained, enabling more practical and scalable applications in hydrological modeling.

Zusammenfassung

Das Verfahren der Bayesschen Inferenz für die Parameter der hydrologischen Modelle spielt eine bedeutende Rolle bei der Ausführung der hydrologischen Modelle, vor allem bezüglich der Genauigkeit und der Zuverlässigkeit. Dieses Thema wird in dieser Arbeit auseinandergesetzt, indem man das Verfahren die Algorithmen zum Markov-Chain-Monte-Carlo-Verfahren implementiert, vor allem mit der Eigenschaft von dem Parallelrechner, damit sich die Genauigkeit und die Effizienz von der Bayesschen Schätzung der Parameter verbessern. Die Ergebnisse bezüglich der Genauigkeit und der Effizienz der Bayesschen Inferenz werden mithilfe der Vergleichsmetriken analysiert, wodurch sie ausführlich visualisiert werden, sodass die Beziehungen zwischen den verschiedenen Implementierungsvarianten und den Ergebnissen klar dargestellt werden können. Außerdem wird die Beziehung zwischen der Wahl der Trainingsdatensätze in Form einer Zeitreihe und die Ergebnisse einer Bayesschen Inferenz beobachtet. Durch die Analyse dieser Aspekte von den spezifischen Implementierungen der Algorithmen und den Datensätzen erwirbt man Kenntnisse über die Leistung und die Ergebnisse der Bayesschen Inferenz, sodass praktische und skalierbare Anwendung bei der hydrologischen Modellierung angewendet werden können.

Contents

Acknowledgements **vii**

Abstract **ix**

Zusammenfassung **xi**

I. Main Part **1**

1. Introduction **2**

2. Introduction to Markov Chain Monte Carlo and Bayesian Inference **4**
2.1. The Idea of Markov Chain Monte Carlo 4
2.2. The Metropolis-Hastings Algorithm 5
2.3. An example of the Metropolis-Hastings Algorithm 8
2.4. Advantages of Markov Chain Monte Carlo Methods over Other Commonly-Used Sampling Methods 10
2.5. The Idea of Parallel Implementation of Markov Chain Monte Carlo 11
2.6. Introduction to Bayesian Inference 11

3. The Use Case of Bayesian Inference in Hydrological Model **13**
3.1. Overview of the Hydrological Model 13
3.2. Overview of the Data Set 14
3.3. Task of This Thesis 14

4. Common Implementation Aspects **18**
4.1. Hardware Specification and Required Frameworks 18
4.2. Evaluation Metrics 18
4.3. Visualization 20

5. Fundamental Implementation **21**
5.1. Basic Metropolis-Hastings and Evaluation Metrics 21
5.2. Knowledge-Based Input Algorithm Parameter Selection 24
5.3. Input Algorithm Parameters Exploration 28
5.3.1. Sampling Out of Bounds 29
5.3.2. Sampling Kernel 35
5.3.3. Likelihood Functions 37
5.3.4. Alternative Implementation of Probability Acceptance Probability . 40
5.3.5. Burn In Phase 41

5.3.6. Effective Sampling Size . . . 43
5.3.7. Iteration . . . 44
5.3.8. Initial States . . . 46
5.4. Result Comparison . . . 47

6. Parallel Metropolis-Hastings 52
6.1. General Idea of Parallel Metropolis-Hastings . . . 52
6.2. Efficiency Analysis . . . 52
6.3. Trace Plot . . . 53
6.4. Gelman Rubin Convergence . . . 58
6.5. Autocorrelation Plot . . . 64
6.6. Accuracy Analysis by the Chains . . . 67
6.7. Parameter Overview . . . 69

7. General Parallel Metropolis-Hastings 73
7.1. Introduction of the Algorithm . . . 73
7.2. Evaluation . . . 74
7.2.1. Ratio Test . . . 74
7.2.2. Amount Test . . . 76
7.2.3. Sampling out of bounds . . . 77
7.2.4. Initial States . . . 79
7.2.5. Dependent Likelihood Kernel Factor . . . 80
7.2.6. The Rest of Input Algorithm Parameters . . . 81
7.3. Parameter Overview and Comparison with the Fundamental Implementation 82

8. The DREAM Algorithm 86
8.1. Algorithm Introduction . . . 86
8.2. Evaluation Based on Chains . . . 90
8.2.1. Efficiency . . . 90
8.2.2. Trace Plot . . . 91
8.2.3. Gelman Rubin Convergence . . . 95
8.2.4. Autocorrelation Plot . . . 99
8.2.5. Accuracy . . . 103
8.2.6. Parameter Overview . . . 105
8.3. Input Algorithm Parameters Exploration . . . 109
8.3.1. Sampling out of Bounds . . . 109
8.3.2. Crossover . . . 110
8.3.3. Likelihood Kernel . . . 113
8.3.4. Initialization . . . 116
8.3.5. Gamma . . . 117
8.3.6. Differential Evolution . . . 119
8.3.7. Burn In Phase . . . 122
8.3.8. Effective Sample Size . . . 123
8.4. Result and Comparison with Other Algorithms . . . 124

9. Sampling Time Series Analysis **127**
9.1. Sampling and Testing Time Series Selection 127
9.2. Result Interpretation 128
9.3. Result Comparison 133

10. Description of Software Implementation **139**
10.1. Structure and Usage 139
10.2. Algorithm Specification 140

11. Conclusion and Further Outlook **143**

II. Appendix **146**

Part I.

Main Part

1. Introduction

The hydrological model HBV-SASK is a computer simulation based on the complex numeric model which is called HBV, also known as Hydrologiska Byråns Vattenbalansavdelning model. Inheriting the core idea of the HBV model which is used to analyze discharge [?], it uses the data from Saskatchewan, Canada, as the name SASK suggests. The model has a few parameters representing the hydrological processes specific to the Saskatchewan region [?]. The uncertainty quantification of these parameters has long been a topic for research, as the parameters, which are important for reliable hydrological predictions, cannot be calibrated exactly. A probabilistic calibration under uncertainty is therefore needed for the model to better reflect local water characteristics and climatic conditions [?]. This is the focus of this thesis, in which the primary objective is to use different approaches to calibrate the input parameters of the hydrology model to determine the posterior distributions of these parameters.

The calibration of these parameters requires a statistic inference method. In this thesis, the Bayesian inference is selected to perform parameters uncertainty quantification, so that the probability distributions of the model parameters can be estimated [?]. The Markov Chain Monte Carlo (MCMC) method is deployed within the Bayesian framework, being used to sample the posterior from the prior distributions of the parameters. This approach allows for a more comprehensive understanding of the parameter uncertainties, leading to more accurate and reliable hydrological predictions [?]. The exact process and algorithm are discussed in the thesis in detail.

The Markov Chain Monte Carlo algorithm uses a Markov chain to perform Monte Carlo simulation, repetitively sampling from the parameter space to approximate the posterior distributions. By running multiple chains in parallel, the computational efficiency and convergence rate of the sampling process can potentially be significantly improved. To investigate this aspect, we emphasized the parallel implementation of the Markov chain Monte Carlo algorithm. In this thesis, the following four versions of Markov chain Monte Carlo algorithms are discussed and used:

- Metropolis-Hastings: The fundamental Metropolis-Hastings method generates the posterior distribution by proposing a new point based on a chosen proposal distribution and accepting or rejecting it with a probability that ensures detailed balance [?].
- Parallel Metropolis-Hastings: The parallel Metropolis-Hastings is a method that is based on the fundamental Metropolis-Hastings algorithm, but uses multiple Markov chains instead of one single Markov chain to run the algorithm in a parallel way to enhance efficiency.
- General Parallel Metropolis-Hastings: The parallel Metropolis-Hastings algorithm extends the fundamental Metropolis-Hastings method by generating multiple samples in one iteration instead of one single sample.

- DREAM: The Differential Evolution Adaptive Metropolis (DREAM) algorithm is an advanced MCMC method that combines differential evolution with adaptive Metropolis sampling to generate the posterior distribution. It deploys multiple chains in parallel and proposes new points using the mutual information between different chains so that the proposal distribution is adaptively updated over time [**?**].

The goal of implementing the four different algorithm versions is to observe the optimized accuracy and efficiency performances of the algorithms. This is achieved by exploring configurations of each of these algorithms are explored. For each algorithm, a set of configurations can be tuned to have an impact on the sampling process of the algorithm. These can be anything from the transition kernel, burn-in factors, and initial states, which will be discussed later in detail. By tuning these configurations, results with different accuracy and efficiency metrics are presented, allowing us to make better decisions regarding how to use the algorithm for uncertainty quantification.

For the last part of the thesis, an overview of the selection of data for the training purpose provided alongside the HBV-SASK model is given. A deep look into the different characteristics and anomaly points throughout the entire time series is provided, allowing readers to gain more insights about the provided data itself. Using different periods with different properties for the Markov chain Monte Carlo algorithms to perform training, the differences regarding the accuracy of the inferred result are analyzed.

2. Introduction to Markov Chain Monte Carlo and Bayesian Inference

In this chapter, the basic theory of Markov chain Monte Carlo algorithms will be introduced, with an example provided alongside visualizations. Afterwards, the idea of Bayesian inference will be explained, including the instantiation of this problem using the Markov chain Monte Carlo algorithm.

2.1. The Idea of Markov Chain Monte Carlo

Markov chain Monte Carlo (abbr. MCMC) is an algorithm that performs sampling. General usage of the Markov chain Monte Carlo algorithm started in the fields of chemistry, biochemistry, and physics up until after 1990 when it was also adopted by the field of statistics and scientific computing [?]. The general idea of the Markov chain Monte Carlo includes, as its name suggests, a combination of the Monte Carlo methods and the usage of Markov chains. The Monte Carlo methods solve numerical problems by repeatedly generating random numbers [?], whereas the Markov chains provide this algorithm a property so that each sample that is generated depends on the sample that is generated before [?].

The Markov chain Monte Carlo tries to sample an unknown target distribution using a proposal distribution that completely lies above the target distribution. The selection proposal distribution is crucial to the success of the algorithm, since it may lead to different behaviors of convergence and acceptance probability [?]. The core of the Markov chain Monte Carlo is the application of a Markov chain, where the Monte Carlo integration is used. Given a distribution $\pi(\cdot)$ and its probability function $f(\cdot)$, a typical Monte Carlo simulation would perform the following mathematical approximation:

$$\mathbb{E}[f(X)] \approx \frac{1}{n}\sum_{i=1}^{n} f(X_i) \tag{2.1}$$

where $\{X_i | i \in [1, n]\}$ is the sample space that is drawn from the distribution $\pi(\cdot)$ [?]. Since this approximation is used on a Markov chain, each sample from the sample space is dependent on the sample before. In Markov chain Monte Carlo, this dependence is given by a transition kernel that is essentially a conditional distribution $\Pr[X_{i+1}|X_i]$ [?]. In other words: After the last sample was generated, a distribution that takes this generated sample as a parameter is created. The next sample is then generated based on this newly created sample, which creates a dependence between both samples. Markov chains Monte Carlo functions thanks to a property called ergodicity that is shared by all Markov chains. The ergodic theorem states that the distribution of the states on the chain converges to a certain stationary distribution regardless of the starting state, as time approaches infinity [?]. This property could be applied in the case of Markov chain Monte Carlo as well. The starting

states that do not sample from the stationary distribution before the ergodic states can be regarded as "burn-in states" and should be discarded. Thus, the approximation of the Monte Carlo simulation can be altered to

$$\mathbb{E}[f(X)] \approx \frac{1}{n-b} \sum_{i=b+1}^{n} f(X_i) \tag{2.2}$$

where b denotes the position of the last state in the burn-in phase that should be discarded [?]. The determination of the burn in phase plays an important part in terms of the sample space accuracy and will be discussed later in this thesis in a more detailed manner.

A crucial prerequisite of the Markov chain Monte Carlo algorithm is the detailed balance condition. The ergodicity property after the burn-in period for every Markov chain can be mathematically defined as:

$$\forall X, Y \in S.\ \pi(X) \Pr[Y|X] = \pi(Y) \Pr[X|Y] \tag{2.3}$$

[?] where S is the set of the state of a Markov chain. From this equation, we can derive the following property:

$$\int \pi(X) \Pr[Y|X] dX = \int \pi(Y) \Pr[X|Y] dX = \pi(X_j) \tag{2.4}$$

What this equation essentially points out is that if X is sampled from the stationary distribution $\pi(\cdot)$, Y will also be from this stationary distribution [?]. This corresponds to the idea of ergodicity and proves that using the detailed balance equation, all of the subsequent samples will eventually connect stationary and target distribution from the very beginning.

2.2. The Metropolis-Hastings Algorithm

Metropolis-Hastings is a widely used algorithm that performs Markov chain Monte Carlo sampling. It is extremely versatile and often used to sample multivariate distribution. It was extensively used in the physics field, but later on also in the statistics field [?].

As mentioned before, the Markov chain Monte Carlo algorithms use a transition kernel to create dependence between two states.

The main idea is that for each iteration, a sample is drawn from the target distribution. For the generation of the next state, a distribution that takes the last sampled data point as a parameter will be created so that the new sample can be drawn from the newly created distribution [?]. An acceptance probability is then calculated using the following formula:

$$\alpha(X, Y) = \min(\frac{\pi(Y) q[Y|X]}{\pi(X) q[X|Y]}, 1) \tag{2.5}$$

where $q[X|Y]$ denotes the proposal density from X to Y. With the proposal density multiplied by the acceptance probability, the transition probability is derived:

$$\Pr[X|Y] = q(X|Y) A(Y, X) \tag{2.6}$$

Due to the detailed balance equation [?]:

$$\pi(X)q(Y|X)A(X,Y) = \pi(Y)q(X|Y)A(Y,X) \tag{2.7}$$

Bringing the ratio of the acceptance probability to the left

$$\frac{A(Y,X)}{A(X,Y)} = \frac{\pi(Y)q[Y|X]}{\pi(X)q[X|Y]} := \alpha(X,Y) \tag{2.8}$$

As we can see, the acceptance probability of the Metropolis-Hastings algorithm is calculated based on the ratio of the acceptance density from the old to the newly generated sample point to the acceptance density from the newly to the old generated sample point. However, since the probability space adds up to one, the maximum acceptance probability can not exceed one. Therefore, a minimum condition must be added.

A special case of the Metropolis-Hastings algorithm is the Metropolis algorithm. The Metropolis algorithm applies a symmetric distribution as the transition kernel, such as a normal distribution [?]. Due to the symmetry, the proposal density $q[Y|X]$ equals to $q[X|Y]$. The idea is that the position of the density of the newly generated sample in the normal distribution centering the last generated sample is at the same altitude as the density of the last generated sample in the normal distribution centering the newly generated sample, as the visualization in Figure 2.1 suggests.

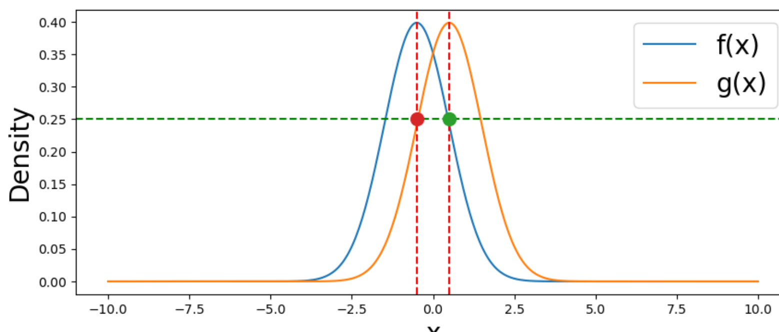


Figure 2.1.: Metropolis-Hastings algorithm with symmetric distribution as transition kernel

In this graph, we can see that the green point is the newly sampled point conditioned on the distribution drawn by the last generated point. The red point denotes, on the contrary, the last generated point conditioned on the distribution drawn by the newly generated sample. These two points lie on the same level and share the equivalent value. Thus, the term $\frac{q[Y|X]}{q[X|Y]}$ cancels out to 1. In this case, the acceptance probability becomes

$$\alpha(X,Y) = \min(\frac{\pi(Y)}{\pi(X)}, 1) \tag{2.9}$$

which is virtually the ratio of the probability density of the newly generated sample to the probability density of the last generated sample. An illustration of this equation would be as follows: if $\pi(Y) > \pi(X)$, which means that the probability density of the newly generated

sample is greater than its of the last generated sample, $\frac{\pi(Y)}{\pi(X)}$ is greater than 1 and the acceptance of the newly generated point is guaranteed. Over time, the samples that are generated will have greater and greater probability density, and eventually, the peak will be reached. If $\pi(X) > \pi(Y)$, which means that the probability density of the newly generated sample is less than it's of the last generated sample, $\frac{\pi(Y)}{\pi(X)}$ is greater than 1 and it is still probable to accept the newly generated point, however a less probability that is calculated equation [**?**].

2.3. An example of the Metropolis-Hastings Algorithm

In this section, an example of the Metropolis-Hastings algorithm will be given and visualized to provide an illustrative comprehension of the different steps of the algorithm. In this example, we want to sample an Erlang distribution from a normal distribution. We are set to generate 100,000 samples to compare how the generated samples fit the Erlang distribution. The Erlang distribution $f(x)$ has a shape of 3 and a scale of 2, whereas the normal distribution $g(x)$ has a mean of 10 and a standard deviation of 1.

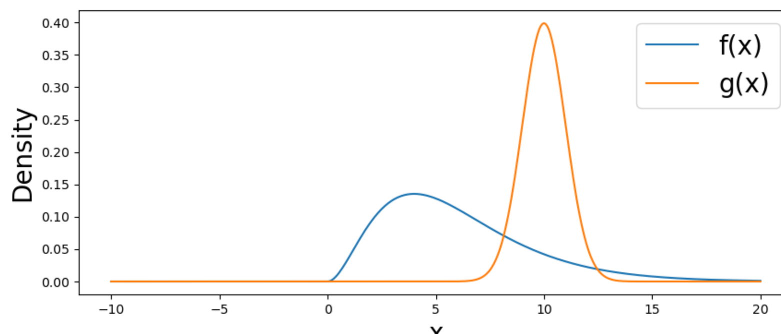


Figure 2.2.: Scaling of the proposal distribution in the Metropolis-Hastings algorithm

As we can see from Figure 2.2, the normal distribution that is sampled does not lie above the Erlang distribution. Therefore, we need to scale up the normal distribution. The scaled up graph is then shown in the Figure 2.3.

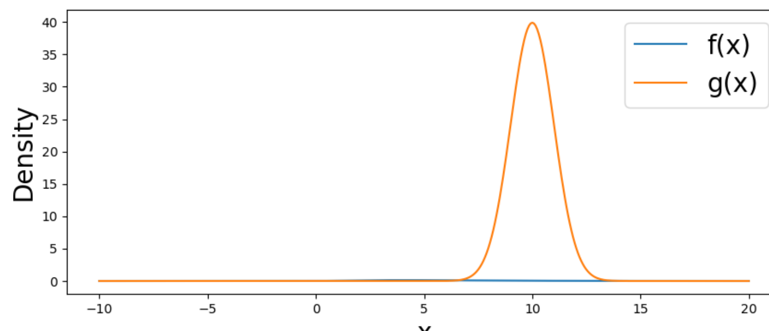


Figure 2.3.: Scaling of the proposal distribution

The transition kernel is set to be a normal distribution that sets the mean as the last generated sample and the standard deviation of 4. Since the normal distribution is symmetric, the acceptance probability could be set to $\alpha(X,Y) = \min(\frac{\pi(Y)}{\pi(X)}, 1)$, as mentioned above. The probability density function of the sampling distribution, $\pi(\cdot)$, is defined as follows:

$$\pi(x) = \frac{1}{\sqrt{2\pi} \cdot \sigma} \exp(-\frac{(x-\mu)^2}{2\sigma^2}) \tag{2.10}$$

where μ denotes the mean and σ denotes the standard deviation of the normal distribution [?].

We select 0 as our starting point and start the iterations from there on. In the first iteration, we acquire the random sample with the value of 1.5437347713886516. The acceptance probability is then $\alpha(X,Y) = \min(\frac{\pi(Y)}{\pi(X)}, 1) = 1$. In this case, since the acceptance probability is 1, it is guaranteed that this sample is accepted. We append this sample to the sample array and move on to the next iteration.

In the next iteration, we sample the value -1.802047900190033. The acceptance probability is then $\alpha(X,Y) = \min(\frac{\pi(Y)}{\pi(X)}, 1) = 0$. This means that this generated sample should by no means be sampled and we should carry on with the last sample. In this case, we repeat our old sample append it to the sample array, and continue.

We continue the rest of the iterations and draw a set of samples. If we plot the samples out and compare them with the target Erlang distribution, it would look something that the graph shown in Figure 2.4. As we can see, the samples that we generate resemble the targeted Erlang distribution. We can conclude that Markov chain Monte Carlo works perfectly in this specific example.

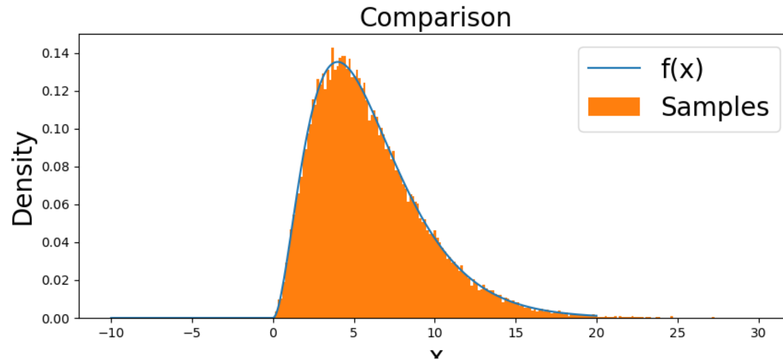


Figure 2.4.: Posterior distribution derived from the Erlang distribution as proposal distribution

2.4. Advantages of Markov Chain Monte Carlo Methods over Other Commonly-Used Sampling Methods

Markov chain Monte Carlo methods differ from other popular sampling methods and have specific advantages over them. In this section, the rejection sampling and the importance sampling are discussed and are compared to Markov chain Monte Carlo sampling.

Rejection sampling, also known as acceptance-rejection sampling, is a sampling method that generates samples that are independent from one another. Instead of generating the next sample from a newly created distribution that takes the last generated sample as input, the samples are generated from a sampling distribution that lies above the target distribution. The acceptance probability is than the ratio of the density of the target distribution over the sampling distribution [?]. However, if the target distribution is complicated, especially in multivariate cases, the ratio might be very low, causing the acceptance probability to be low as well, which leads to inefficiency. By creating a dependency between samples, Markov chain Monte Carlo methods avoid this inefficiency [?].

Importance sampling is a sampling method that is based on the calculation of weights. A typical implementation includes the generation of samples, calculating the weights of all of these samples, and calculating the expected value by summing them up together by the weights [?]. The process of importance sampling is rather easy to implement. However, a disadvantage is that the samples that have higher weights dominate the calculation of the expected value, which essentially reduces the sample space since the samples that have lower weights play almost no role in the calculation. Markov chain Monte Carlo methods, on the other hand, eventually samples from the stationary distribution after the burn-in period, resulting in consistency of the sampling [?].

2.5. The Idea of Parallel Implementation of Markov Chain Monte Carlo

The objective of this thesis is to implement parallel versions of Markov chain Monte Carlo algorithms for the Bayesian inverse problem. Therefore, paralleling Markov chain Monte

Carlo is a significant part of the thesis. The base idea is that instead of one single chain, several chains are run simultaneously [?]. Since after the burn-in period, every single state from each chain samples from the stationary distribution due to the ergodic property of the Markov chains [?], all of the samples from different chains could be merged to present the target stationary distribution. However, other variations involve generating multiple points at the same time conditioned on the last generated sample and then evaluating the forward model [?]. Since this chapter only provides an overview of the Markov chain Monte Carlo algorithm, a following chapter that is specifically dedicated to the parallel of the algorithm will be given.

2.6. Introduction to Bayesian Inference

The Bayesian inference problem is a method of statistical inference that is used to calculate the probability estimate based on evidence and the likelihood of the set of parameters [?]. Given a prior distribution that provides information on the preexisting data, the Bayesian inference problem uses the Bayes theorem to update the prior distribution using a likelihood function and derives the actual possibility.

Given the Bayes theorem [?]:

$$\Pr[B|A] = \frac{\Pr[B] \cdot \Pr[A|B]}{\Pr[A]} \tag{2.11}$$

where A and B are two different incidents. This equation can than be formed into

$$\Pr[B|A] = \frac{\Pr[B] \cdot \Pr[A|B]}{\int \Pr[B] \cdot \Pr[A|B] dB} \tag{2.12}$$

$\Pr[A|B]$ is called the likelihood function, which is generated by a set of data to interpret how likely a particular set of observations is [?]. The $\Pr[B]$ is called prior, since this is the preexisting knowledge that is given [?]. The denominator of the equation is called evidence, which is a constant that depicts the probability of observing the data across all values of the model parameters [?]. The result of the above equation is the posterior, which is the object of the Bayesian inference problem [?].

The different implementations of the Bayesian inference problem are versatile and vary from one another. In this thesis, we implement the Bayesian inference problem using the Metropolis-Hastings algorithm with a normal distribution transition kernel. The idea is that we calculate the acceptance probability based on the posterior calculation. For revision, the acceptance probability of the Metropolis-Hastings algorithm with a normal distribution transition kernel is given by [?]:

$$\alpha(X, Y) = \min(\frac{\pi(Y)}{\pi(X)}, 1) \tag{2.13}$$

Replacing $\pi(\cdot)$ with the posterior distribution, we derive:

$$\alpha(X, Y) = \min(\frac{\Pr[Y] \Pr[X|Y]}{\Pr[X] \Pr[Y|X]}, 1) \tag{2.14}$$

In plain language: The acceptance probability is calculated by the ratio of the prior and likelihood of the newly proposed point over the prior and likelihood of the last generated point. Since the evidence is a constant, they cancel each other out and will therefore not be taken into account.

Different variants of implementations are used throughout this thesis to perform Bayesian inference of the hydrological model. They will be discussed in later chapters. For now, we will take a look at the hydrological model.

3. The Use Case of Bayesian Inference in Hydrological Model

In this chapter, we focus on the hydrological model that is used in this thesis. An overview will be given, followed by the explanations of hyperparameters. In the last section, the dataset used in this thesis is observed and visualized to provide a practical understanding for the hydrological model.

3.1. Overview of the Hydrological Model

The HBV-SASK conceptual model is a renowned mathematical model that is commonly used in the field of hydrology. HBV is a model that describes the subroutines for snow accumulation and melts, for soil moisture accounting and river routing [?]. SASK stands for the province of Saskatchewan, the province in Canada in which the model is developed. The creation of the HBV-SASK model is therefore based on the HBV model but involves local data calibration and integration with local water management needs [?].

The HBV-SASK model has twelve different hyperparameters, of which seven are relevant to this thesis [?]. These include [?]:

- TT: Ranges from -4 to 4. It stands for the air temperature threshold in °C for melting/freezing and separating rain and snow.
- C0: Ranges from 0 to 10. It describes the base melt factor in mm/°C per day.
- β (beta): Ranges from 0 to 3. It depicts the shape parameter (exponent) for the soil release equation.
- ETF: Ranges from 0 to 1. It describes the temperature anomaly correction in 1/°C of potential evapotranspiration.
- FC: Ranges from 50 to 500. It depicts the field capacity of soil in mm.
- FRAC: Ranges from 0.1 to 0.9. It stands for the fraction of soil release entering the fast reservoir.
- K2: The slow reservoir coefficient ranges from 0 to 0.05, which determines what proportion of the storage is released per day.

To run this model, these hyperparameters need to be determined. Since the only prior information given is the lowest and the highest bound of each hyperparameter, uncertainty quantification of these hyperparameters is, therefore, necessary to gain posterior information. Apart from these hyperparameters, the starting and the end date of the period that is used for uncertainty quantification are also required to be specified [?]. However, the very first phase at the start is used for the spin-up phase, in which the model runs for some time

using historical data. This phase stabilizes internal model states such as soil moisture and groundwater levels, which are important for accurate simulation [?].

3.2. Overview of the Data Set

There are two existing data sets to the hydrological model, which are respectively called Oldman Basin and Banff Basin, since they are each measured at the Oldman River and in the town of Banff in Alberta, Canada [?]. The value that is measured is called streamflow. It describes the movement of water within a river or stream channel and is the combined result of all climatological and geographical factors that operate in a drainage basin [?]. Both of these data sets are presented in the format of a time series, in which the value of each measurement is collected against the dates over a long period. The Oldman basin data set is available from 1979 to 2008, whereas the Banff basin data set is available from 1950 to 2011 [?].

Since the data is presented in a format of time series, a time series decomposition is required to present more information on trends, and seasonal and regression effects [?]. After the decomposition, the trend, seasonal, and residue of the dataset over the whole period can be observed. For the time series decomposition in this section, the function TSA.seasonal.seasonal_decompose from the python framework stats models is used [?].

First, we take a look at the result of the decomposition of the Oldman basin data set. It is shown in Figure 3.1. The seasonal component of this time series is regular over the years. However, there were significant peaks around the early 1980s, mid-1990s, and around 2005. The streamflow in these periods is higher than anytime else, which indicates that there might be possible heavy rainfalls, floods [?], or even ecologic disasters. Anomalies in this period can be also found in the residue component, which confirms that there might be odd behaviors happening in these periods. Therefore, quantifying the uncertainty of these three periods would be a challenging task.

We then take a look at the result of the decomposition of the Banff basin data set. It is shown in Figure 3.2. Similar to the case of Oldman Basin, it shows a regular seasonal component. The trend, on the other hand, fluctuates across the entire measured period but does not show a general upward or downward direction. This means, that the whole measurement is indeed relatively stable. However, the fluctuation might still impose an influence on the accuracy and stability of uncertainty quantification. The residuals of the Banff basin data set are much more varied than the Oldman basin data set. There are obvious clusters of high activity and anomalies, which confirms that the uncertainty quantification of this data set might be a hard task to deal with.

3.3. Task of This Thesis

Since the task of this thesis is to caliber the input algorithm parameters of the hydrology model, the ultimate goal is to acquire the posterior of the parameters. The main idea in this use case is that we infer the posterior probability of each parameter using Markov chain Monte Carlo. The exact process goes as follows: First, a model is created with the configuration of the period and target distributions being given. Then, we select an appropriate Markov chain Monte Carlo algorithm and perform sampling to solve the Bayesian

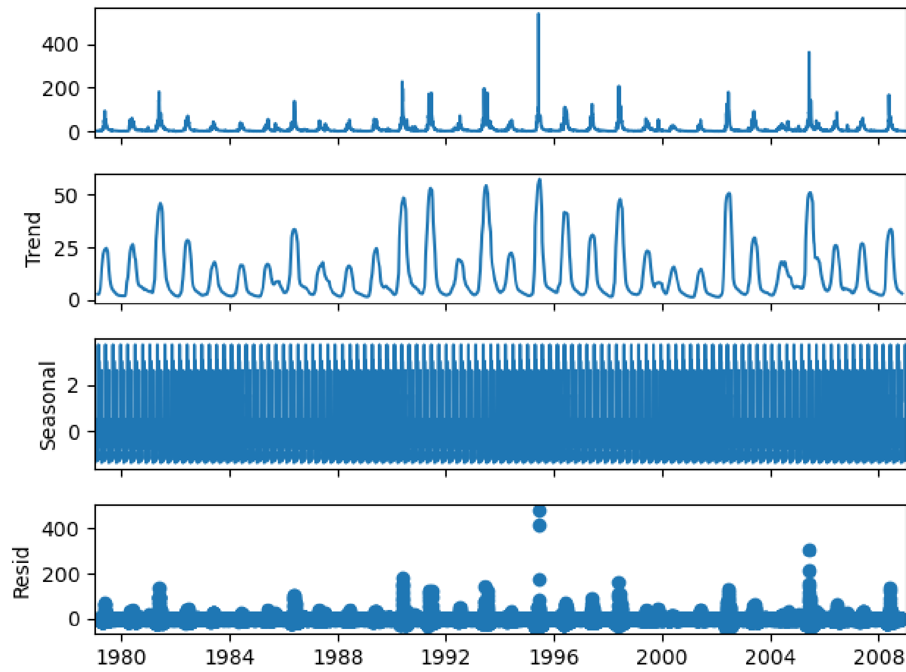


Figure 3.1.: Time series decomposition of the Oldman dataset

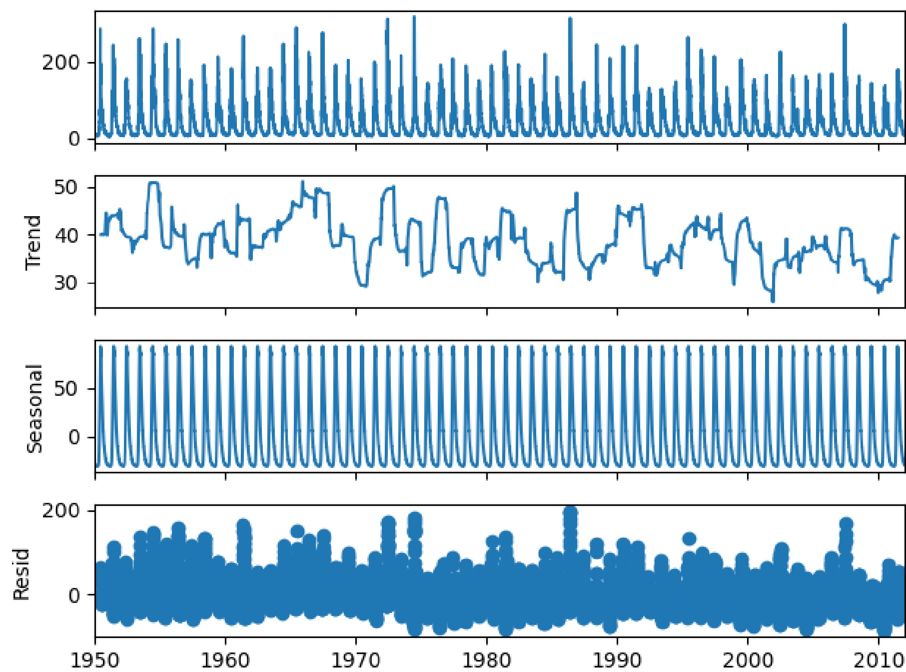


Figure 3.2.: Time series decomposition of the Banff dataset

inference. In this process, the output of each generated set of parameter arrays will be calculated by the model. The output will be later compared to the measured data points, which will provide the algorithm with an acceptance probability. In each iteration, the algorithm decides whether to accept or reject this sample. After all these iterations, the posterior probability is obtained and is prepared to be used on a testing data. Last but not least, we compare the projected results that are calculated by the model to the measured result. Since the posterior probability that is obtained is denoted in the form of a list of sampled data, we can use the Monte Carlo simulation to generate a certain amount of samples, pass them into the model and acquire the result. Then, we can either calculate the mean of the results or sample the maximum of the result to compare them with the actual data.

In order to operate the training and the testing data periods are needed to be determined and extracted. For the Oldman Basin, the trend of the data from 1990 until 2000 remains constantly fluctuated within a specific upper bound and a specific lower bound, which indicates the stability of the record. Therefore, the posterior is inferred by the time period of the entire 90s. An additional ten years beforehand, which is equivalent to the time period on which the posterior is trained, is used for the spin up phase. The time period after 2000 is then used as the testing phase. Since there are some anomalies around 2005, any time period that includes the year 2005 would be an excellent time period for testing. For the Banff Basin data set, the recorded data is kept constant after the year 1970 and the year 1990. However, in order to use the data in the 2000s for testing purpose, the data before 1990s does not play such an important role. Therefore, we use the data in the time period of the 1990s for training purpose, whereas the 10 years beforehand are used for the spin up phase. The entire time period after the 2000s would be excellent for testing purpose, since anomalies and extreme values can be found throughout the entire sequence.

To test the generalized application of the algorithm, we select three challenging yet representative periods for testing, with one being long (5 years), another being medium long (around 2 years) and the other being short (1 year). Due to the constant fluctuation of the Banff basin data set, we could select a long period within it to perform uncertainty quantification. Since the residue in the early 1970s seems to show extreme anomalies, we select 1970 to 1974. The other two periods are both picked from the Oldman Basin data set. The period from 1994 to 1996 is chosen, since there are apparent anomalies of the residue in this period. Another period is the year 2005, since there are also anomalies being showcased in this year.

4. Common Implementation Aspects

In the next four chapters, four different algorithms of Markov chain Monte Carlo are discussed in this thesis. Different setups are going to be tested for different algorithms so that the gathered results can be analyzed. However, certain common aspects are shared among the background and analysis of all these four algorithms. These are detailed in this chapter.

4.1. Hardware Specification and Required Frameworks

All of the code that is run and tested in this thesis is run on a single computation machine, namely the MacBook Pro 2021 by Apple Inc. It has an Apple M1 Pro chip, which has an ARM architecture. It has 10 CPU cores, 8 of which are for performance and the rest are for efficiency. It has a RAM of 32 GB and also 16 GPUs available[1]. The entire code is run on macOS Sonama 14.4.1.

The software implementation of the fundamental Metropolis-Hastings algorithm is rather basic. Since the sample space is multivariate, the calculation involves operation between vectors. To ease the process of these calculations, the popular software package of Numpy is used [?]. Additionally, the Tensorflow Probability package is used. It does not only offer implemented probability distribution functions, but also randomness and sampling functions [?]. To use the Tensorflow Probability framework, the software package Tensorflow is required to be installed[2]. For the visualization part, standard data visualization libraries including Matplotlib and Seaborn are used for the creation of histograms, kernel density estimation, and boxplots [?].

4.2. Evaluation Metrics

To determine which configuration of the algorithm delivers better results, evaluation metrics need to be set up. All of the algorithms run in this thesis are evaluated in accuracy and efficiency. The accuracy measures how closely the outcome of the algorithm aligns with the actual measured data, whereas the efficiency keeps track of the run time of each algorithm run.

To quantify the accuracy, we need to introduce metrics that can calculate the accuracy of the Bayesian inference results. First, we calculate the mean of the absolute difference between the calculated time series run by the model and the measured time series. By calculating the absolute difference, the similarity of the times series could be well quantified. Afterward, two metrics that are used to test the goodness of fit are calculated, namely root mean square error (RMSE) and mean absolute error (MAE). The RMSE is a metric that is

[1] `https://support.apple.com/en-us/111902`
[2] `https://www.tensorflow.org/probability`

often used to evaluate the model performance in climate research studies and ecology. The calculation is as follows:

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2} \tag{4.1}$$

The square root in RMSE plays an important role. On the one hand, RMSE penalizes larger discrepancies more severely by squaring the errors [?]. Moreover, it helps to stabilize the variance of the error terms, making it particularly useful in the use case of this thesis, since the usage of standard deviation and variances are present in the implementation [?].

The MAE is another widely used metric for model performance evaluation [?].

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| \tag{4.2}$$

Calculating the average of absolute errors means that it is easier to understand and interpret the calculation directly. Besides, due to the less impact of anomalies on the metric [?], MAE can offer a more robust estimate in contexts when extreme values are expected to be anomalies.

The whole process of evaluation of the results is going to look like this. To collect this result, we first randomly generate 1000 samples from the given posterior using the Monte Carlo simulation [?] and take the mean. The results of the calculation are saved as "posterior mean". Calculating the mean value, which generalizes the entire results, would allow us to observe the actual accuracy of the result since every single individual time series contributes to the calculation process. Next, the maximum value of each timestamp is also found, so that we can observe how extreme the posterior samples could be. The result is then saved as "posterior max", which would indicate how stable the sampling process is. If the posterior max does not differ much from the posterior mean, then the entire sampling space is stable. Otherwise, the individual samples vary too much from each other, causing destabilization. These two time series are then compared to the "prior mean" time series, which is the mean of all the calculated time series of the model that takes samples from prior as input, to observe how much the prior distribution influences the result. After calculating all of the above, a visualization is going to be done so that these results can be compared to each other.

For simplicity, we only use the Oldman Basin dataset for the algorithm to perform sampling. For the exploration phase, the model will be run and evaluated on the same training dataset, so that we can observe how well the trained posterior fits the data on which it is trained. For the actual evaluation, the model will be run on the training dataset, whereas the accuracy score will be evaluated from a testing data set. The year 2005 of the Oldman Basin dataset is selected, not only because the data is recent enough, but also because the data contain both calm periods and anomalies.

Apart from the accuracy test, efficiency also plays an important role in the MCMC algorithms [?]. To test the efficiency, the run times of different implementations are going to be recorded, so that a comparison can be done later and we can infer which factors have impacts on the computation time.

4.3. Visualization

Visualizations directly improve our understanding of the results and play an indispensable role in this thesis, where different plotting mechanisms are applied. In this section, all of the plots that are used in this thesis are described.

For the use case of the Bayesian inference problem, two types of visualizations are necessary. First, we need to know how the individual parameter distributes in the posterior probability. The result of the Metropolis-Hastings is a multidimensional list of values, representing the calculated posterior. For each parameter, a separate histogram is generated. Alongside the histogram is the Kernel Density Estimation (KDE) graph. It is used to estimate the probability density function of a random variable based on a data sample by averaging contributions from specific kernel functions that are centered at each data point [**?**]. By combining the histogram and the KDE graph, an overview of the general distribution of the posterior can be understood.

However, if the histogram and the KDE plot resemble the prior distribution, little or no useful information can be retrieved. Therefore, we can visualize the data using boxplots to get specific information on the distributions of the posterior of each parameter. The data we can read from the boxplot are the locations of different samples so that we can know how many samples are located under the first quantile, above the third quantile, or the median. It provides us with an easier understanding by explaining the locations of all of the samples in the posterior distribution.

At the very end of the parameters visualization, the calculated results are going to be compared. The four times series including posterior mean, posterior max, prior mean and measured data, which are mentioned before, are going to be plotted on the same graph so that the results can be visually compared.

The other type of visualization is presented during the exploration phase of the input algorithm parameters. The charts present the relations between the input algorithm parameter configurations and the metrics results. If the configuration is the form of numeric values, the line plot is drawn, so that a possible existing trend could be detected. If the configuration contains categorical values, then a bar chart is drawn instead. By plotting the metrics against the configuration, a clear comparison of the results calculated by different sets of input algorithm parameters is provided.

5. Fundamental Implementation

In this chapter, a basic version of Metropolis-Hastings is implemented. Complications including the parallel aspect, more complicated algorithms, and further enhancement are not considered in this chapter. A general idea of the implementation is explained, with specific sections contributing to possible issues with the general algorithm and the choice of the likelihood function.

5.1. Basic Metropolis-Hastings and Evaluation Metrics

First of all, we take a look at the basic implementation of the Metropolis-Hastings algorithm. This implementation applies the idea of the algorithm mentioned in the chapter above. It takes the following required input algorithm parameters: the proposal distribution that data points are sampled from, the sampling kernel for transition, the likelihood kernel for calculation of the likelihood and acceptance probability, the initial state where the algorithm starts, and the number of iterations. Since the prior distributions of the parameters that are going to be calibrated are uniformly distributed, the boundaries need to be given as input algorithm parameters, so that the algorithm can examine whether the generated samples are out of bounds, which should be correctly handled. The sampling kernel that is used throughout this thesis is the Gaussian normal distribution since it provides symmetry and simplicity [?]. The concrete algorithm is listed below.

Algorithm 1: Basic Metropolis-Hastings Algorithm

Input: proposal distribution function, sampling kernel function, likelihood kernel function, initial state, number of iterations

Output: list of sampled data points

```
1  Function MH(proposal_dist, sampl_kernel, likel_kernel, init_state, iterations):
       // Initialize the samples list with the initial state
2      samples ← [init_state]
3      for i ← 1 to iterations do
           // Generate a new sample from the sampling kernel
4          old ← samples[i-1]
5          new ← sampl_kernel(old)
           // Calculate the acceptance probability
6          acceptance_ratio ← proposal_dist(new)
             ·likel_kernel(new) / (proposal_dist(old)·likel_kernel(old))
7          acceptance_probability ← min(1, acceptance_ratio)
           // Decide to accept or reject the new sample
8          if random() ≤ acceptance_probability then
9              samples.append(new)
10         else
11             samples.append(old)
12     return samples
```

There are two required kernels for the input algorithm parameters, namely the sampling kernel and the likelihood kernel. The sampling kernel, as mentioned above, acts like the transition of the Markov chain between two consecutive samples. It plays a critical role in the performance and results of a Markov chain Monte Carlo simulation [?]. Due to the ease and efficiency of calculation that is discussed above, symmetric distribution, notably normal distribution, is used in this thesis [?]. However, the shape of the kernel is left unknown and is to be determined by the standard deviation [?]. The choice of the standard deviation is crucial. If the standard deviation is too wide, the generated samples will keep getting out of bounds. If the standard deviation is too narrow, it will take too long time to explore the whole distribution range, because it is more likely that the new sample generated is near the mean.

The likelihood kernel also plays a crucial role in the Markov chain Monte Carlo. It measures the probability of observing the data given the set of the generated parameters under the model [?]. In this thesis, the likelihood function for the kernel is implemented as a normal distribution probability function due to the Central Limit Theorem, which suggests that the mean of a large amount of independent random variables will be approximately normally distributed regardless of the underlying distribution [?]. In other words, the normal distribution describes the noise of the data points around the mean, which is the sampled value. The likelihood function of the Gaussian normal distribution is given by the product of all of the probability densities of each point [?]:

$$L(\mu, \sigma^2; x_1, ..., x_n) = \prod_{j=1}^{n} \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x_j-\mu)^2}{2\sigma^2}} \tag{5.1}$$

Given that the calculation process is numerically hard to solve, the log-likelihood function is commonly used [?]. The log-likelihood function of the Gaussian normal distribution is given by [?]:

$$\log[L(\mu, \sigma^2; x_1, ..., x_n)] = -\frac{n}{2} \ln(2\pi\sigma^2) - \frac{1}{2\sigma^2} \sum_{j=1}^{n} (x_j - \mu)^2 \tag{5.2}$$

Both of these above equations are implemented as help functions in this thesis for easier further applications. However, an easier way of implementation is to use a probability modeling framework to create a multivariate normal distribution centered around the given value, calculate the likelihood for each pair of points, and sum them up together. This version is also implemented using the TensorFlow probability library and is mainly used throughout this thesis. The only unknown thing is the standard deviation of the distribution. Similar to the sampling distribution, the standard deviation is a left-to-be-determined variable that describes the shape of the distribution. In the context of the likelihood function, the standard deviation implies how concentrated the data are expected to be around the mean value [?]. A smaller standard deviation suggests that the data points are expected to cluster tightly around the mean and have less tolerance, whereas a larger standard deviation implies a broader spread of data points and more tolerance.

Using the above-listed algorithm, we can perform Bayesian inference on the HBV-SASK model. A problem to consider here is the acceptance probability calculation. The joint probability of a seven-dimensional uniformly distributed parameter space is the product of

all of the probabilities of each dimension, making the joint probability density to always be a tiny number. In this case, the sample generated are likely to be rejected in every iteration, which causes a acceptance probability that is closed to zero and does not allow the algorithm to explore the entire parameter space. Therefore, an alternative way of calculating the acceptance probability is needed.

One way of implementing an alternative is to take the mean of the probability distributions across all dimensions as the final acceptance probability. For one, individual acceptance probabilities can vary widely while handling the joint distribution. Averaging these rates can give a more stable estimate of the overall acceptance behavior, making sure that the acceptance probability is appropriate and meaningful. For another, equal contribution to the proportional rate of each dimension is ensured, thus allowing the acceptance probability to represent the generality of each dimension. Another option would be taking the maximum value instead of the mean from the acceptance probabilities of each dimension, which ensures that at least one dimension is being adequately sampled. Through the maximum selection of the acceptance probability, sufficient attention to dimensions that are harder to sample than others is also paid, so that the sampling efficiency would be higher than other ways of implementation. Both variants of implementation are discussed later on in detail.

Another thing that needs to be done is to select meaningful values or instances to be input algorithm parameters. In the following sections, we are going to discuss the choices of input algorithm parameters based on two methods. First, the input algorithm parameters are going to be selected using existing knowledge. Afterward, the input algorithm parameters are going to be explored, in which models with different input values are going to be run multiple times. Afterward, the accuracy tests that are mentioned above are going to be carried out, so that an overview of the relation between different input algorithm parameters and the accuracy and efficiency result can be visualized. An overview of which set of input algorithm parameters delivers the best overall accuracy and efficiency is also going to be shown later on. To test how well the posterior fits the data on which it is trained, the evaluation part is going to be carried on the training data as well. In the end, the sets of input algorithm parameters from both methods are going to be used for another run of the algorithm, so that the results of the Bayesian inference problems can be compared with each other. The evaluation, however, is going to be carried on on the testing data, so that we can observe how well the posterior would perform in actual cases.

5.2. Knowledge-Based Input Algorithm Parameter Selection

Before exploring the model's input algorithm parameters, we can determine the input algorithm parameters by logic previous knowledge, and logical thinking. In this section, the Bayesian inference problem will be directly executed, as well as the visualization part. First, all different input algorithm parameters are going to be individually discussed.

The proposal distribution is, for the use case of the hydrological data set, relatively straightforward. Since there is no further information regarding the shape and look of the distribution other than the upper and the lower bound, a multivariate uniform distribution that ranges from the lower and the upper bound could be modeled and used as the proposal distribution.

The determination of the standard deviation of the sampling kernel is crucial to the result

of the algorithm. Since the standard deviation should generally not be a bigger value than $\frac{1}{4}$ of the range [?], we start to find a maximum value of the factor going down from there so that the transition kernel of the algorithm does not go out of bounds too often, which causes numerical errors. After several test runs of the model, we found out that the first appropriate sampling kernel is a normal distribution with the standard deviation set as $\frac{1}{6}$ of the range, since it is neither too wide nor too narrow, which allows the algorithm to run smoothly and effectively reduce the amount of sampling out of bounds. This value is thus set as the default standard deviation.

The default standard deviation value is set to 1. The intention of setting the standard deviation to a relatively low value is to expect better precision. Since normal distributions with narrow standard deviations give out lower probabilities if the value is away from the mean, the samples that are far away from the mean will receive more penalties, if the standard deviation is set low. This results in a lower likelihood value, which might lead to a lower acceptance probability.

For the rest of the attributes: The initial state does not affect the accuracy of the result [?]. Therefore, it is set as a random set of values within the uniform distribution for now. To increase the random effect, we generate 1000 random samples and take the mean as the random result. Optimization for efficiency is going to be performed later in this thesis. For now, we simulate the Bayesian inference problem using Markov chain Monte Carlo for 10.000 iterations. As suggested, 20 percent of the data are discarded due to the burn-in phase [?].

We execute the algorithm with the above-suggested input algorithm parameters. The execution was successful and produced decent results. Several graphs are produced to provide a great visualization of the result.

First, we take a look into the posterior distribution of individual parameters. The graph is shown in Figure 5.1. As we can see from the histogram and the KDE plot, the calculated posterior does not resemble specific distributions. They share a similarity, that is the probability of samples near the boundaries are relatively lower than the samples near the center. All of these parameters also have several peak values, which are sampled in the posterior distribution more than other values. All of the values are relatively widespread and evenly distributed.

Since little information is gathered from the above plot, we can visualize the data using boxplots to get specific information on the distributions of the posterior of each parameter. As we can see from the boxplots that are shown in Figure 5.2, the boundaries of the posterior distributions are retained. They share the same boundaries as the prior distribution. However, the first and third quantile as well as the median of the posterior distribution. This gives us a general idea of how the posterior would look like. For further application of Markov chain Monte Carlo algorithms, the starting values could potentially be altered based on this information to improve efficiency. This aspect will be discussed later in this chapter.

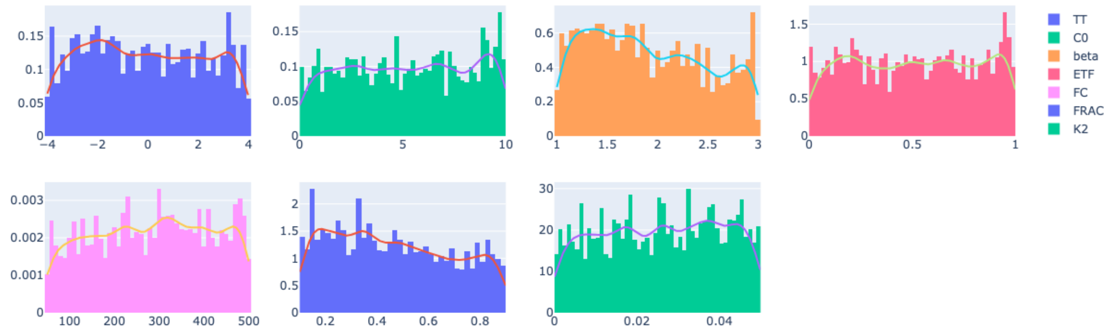


Figure 5.1.: Overview of the posterior distribution of the parameters calibrated by the default Metropolis-Hastings algorithm

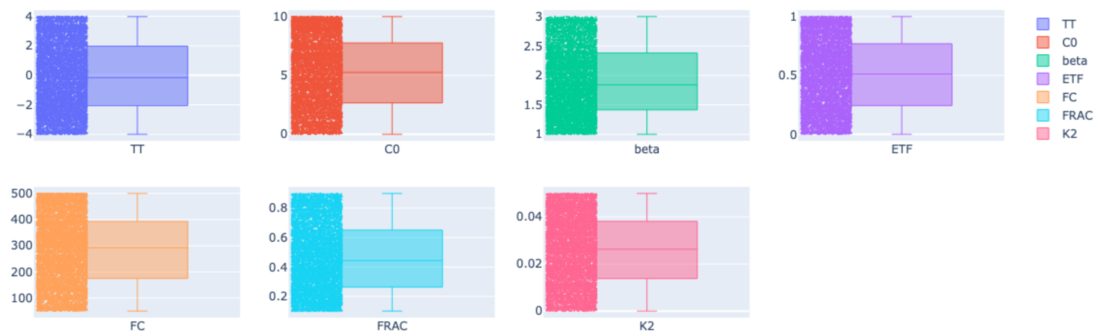


Figure 5.2.: Boxplots of the generated posterior samples of each parameter calibrated by the default Metropolis-Hastings algorithm

After analyzing the parameters individually and as a group, the result that is computed by the posterior of the Bayesian inference is revealed in Figure 5.3 and compared to the observed data. The calculated result resembles the actual measured data, particularly the posterior mean. It reaches its peak in the same period as the measured data, whereas it shows stable behavior for the rest of the time, just like the the measured data. The posterior max shows slightly more extreme behaviors at certain points. Both of the posterior time series show significant improvement based on the prior mean.

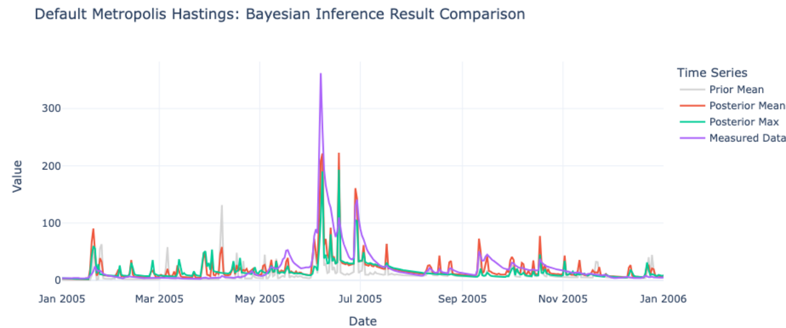


Figure 5.3.: Comparison of Bayesian inference results of the default Metropolis-Hastings

After visualizing the parameters, we take a look at the accuracy of the data. For this part, the RMSEs and MAEs are calculated. The RMSE of the posterior mean is 22.14475485613421 and the MAE of the posterior mean is 11.399487080387862. From the graph, we can see that a significant difference in the results is shown around the peak, which might contribute to the relatively high MAE and a decent value of RMSE. The RMSE of the posterior max is 26.72454579307846 and the MAE of the posterior max is 13.196081237340492. They do not show too many differences from the posterior means, which means that the posterior distribution is relatively stable.

These values will be later compared to the results after the input algorithm parameters exploration. In the following sections, we are going to explore specific parameters of the Metropolis-Hastings algorithm to achieve maximum accuracy. Besides, the efficiency will also be taken into account, where the run time of the algorithm will be observed and compared.

5.3. Input Algorithm Parameters Exploration

After finding the appropriate values by knowledge-based selection, all of the input algorithm parameters will be explored in this section. By trying out different reasonable values as input and interpreting the accuracy and efficiency results, we might be able to figure out a specific relation between the different input values against the accuracy or the efficiency score. Later on, the algorithm will be run using the set of input algorithm parameters which delivers the best performance by accuracy and efficiency metrics and be compared with the input algorithm parameters by knowledge-based selection on the testing data.

5.3.1. Sampling Out of Bounds

While executing this algorithm for the hydrological model, however, there is a certain issue. Since no specific information regarding the distribution is given, we are required to use the uniform distribution to describe the parameters that need to be calibrated. Since the uniform distribution ranges from a certain lower bound to a certain upper bound, it does not have an unlimited range. In this case, there is a possibility that the newly generated samples are out of bounds, which is not helpful for the calibration. For example, if a generated point,

which is accepted, is out of bounds, the p variable in the algorithm will be set to 0, which causes invalid values like negative infinity to occur in the calculation while using logarithm likelihood functions. If these values are sampled and carried on, values that are further from the bounds may be going to be sampled, which leads to mistakes in the result. Therefore, measures need to be taken to avoid these errors from happening. Three implementation variants against this issue have come up. These are ignoring, boundary aggregation, and reflecting boundary.

The ignoring method is the default method and the most straightforward: Any points that are generated outside of the bound are going to be eliminated. It is extremely important to mention that instead of taking another sample from the iteration, the last generated sample needs to be carried on. Since the sample out of bounds is supposed to be an impossible case, the acceptance probability in that point needs to be treated as 0, which means that this point is directly rejected. For multivariate distribution, a set of data points needs to be rejected entirely at once if one single data point is out of bounds since the acceptance probability otherwise would be different. Otherwise, the detailed balanced condition could be violated, as the transition probabilities would not be symmetric anymore for every single parameter.

The boundary aggregation method is different in treating the out-of-bounds samples, in which it transforms the sampled data that are out of bounds. We simply sample the upper bound or the lower bound and carry on from there. If a new sample is generated based on the normal distribution that is centered around the out-of-bounds sample, there is less or equal to fifty percent chance that the newly generated point is inside the range [**?**]. The main benefit of this method is that there is still a minimum of fifty percent chance that the next sample is kept inside the bound in cases where the samples are out of bounds. For multivariate distribution, a single data point in each dimension can be handled individually, however, the acceptance probability needs to be calculated based on the transformed sample points.

The posterior that is derived from the boundary aggregation method is shown in Figure 5.4. A few samples are getting aggregated on both sides of the boundaries. If we ignore these two bars on both sides, the distributions of the rest of the samples of each parameter still resemble uniform distributions, which does not provide a lot of information regarding the actual distribution of the parameter.

The boxplot of these samples is shown in the Figure 5.5. An obvious takeaway from this chart is that all of the boxplots have a lower first quantile and a higher third quantile, whereas the median is retained almost at the same place. The disposition of both of these quantiles is the result of the aggregation of samples on both sides of the boundaries.

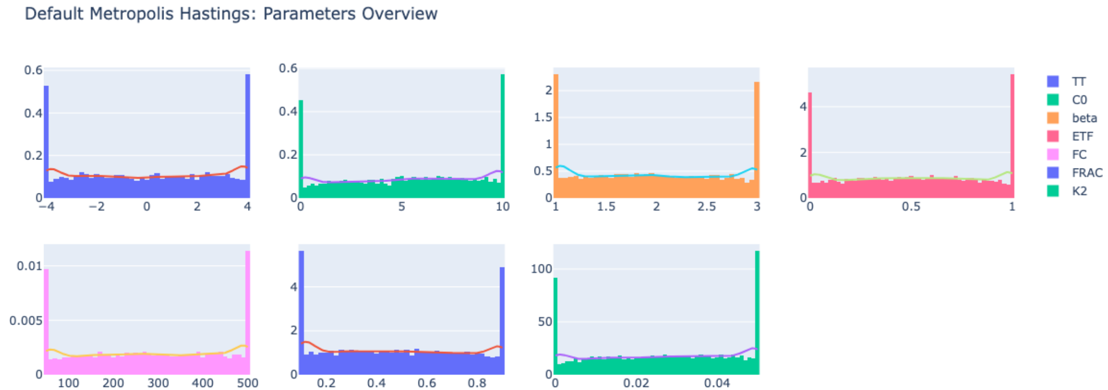


Figure 5.4.: Overview of the posterior distribution of the parameters calibrated by the Metropolis-Hastings algorithm that samples the bound value if the sample is out of bounds

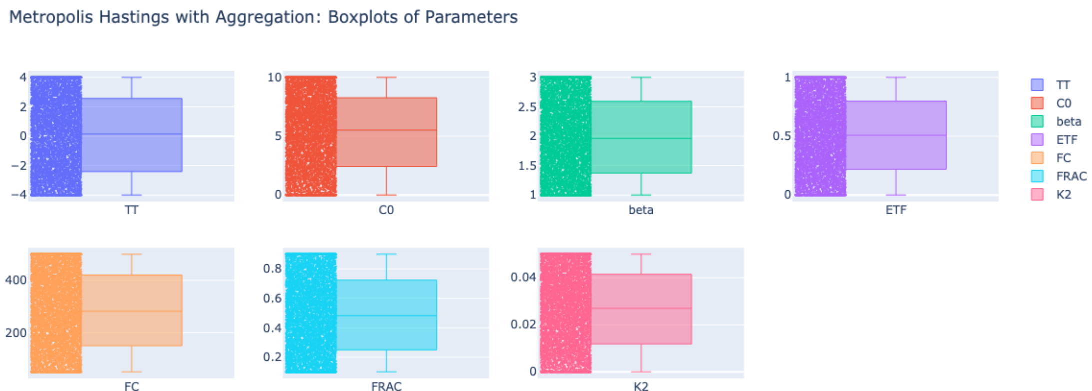


Figure 5.5.: Boxplots of the generated posterior samples of each parameter calibrated by the Metropolis-Hastings algorithm that samples the bound value if the sample is out of bounds

The reflect boundary method also transforms the data instead of ignoring it but in a different way than the aggregation method. Due to the symmetry of the transition kernel, the possibility of sampling an out-of-bounds sample is the same as the possibility of sampling the point symmetric over the mean of the Gaussian normal transition kernel [**?**]. In this case, the acceptance probability of the transformed sample point is not changed, which has no interference with the algorithm itself and future samples. For multivariate distribution, a single data point in each dimension can be handled individually and the acceptance probability does not need to be recalculated due to the symmetry of the transition kernel distribution.

We now take a look at the posterior distribution of the samples. It is shown in Figure 5.6. From the graph, we can see that the posterior distributions sampled from this version of Metropolis-Hastings look very much different from the others. All of these posterior distributions resemble normal distributions, with one peak somewhere in the middle. For some parameters like C0, beta, ETF, and TT, the boundary is shifted, which means that no samples from the region near the boundaries are generated. Since all of the samples that are out of bounds are reflected, the reflected samples compensate the holes of the non-reflected samples, so that it gives rise to a normal distribution like posterior.

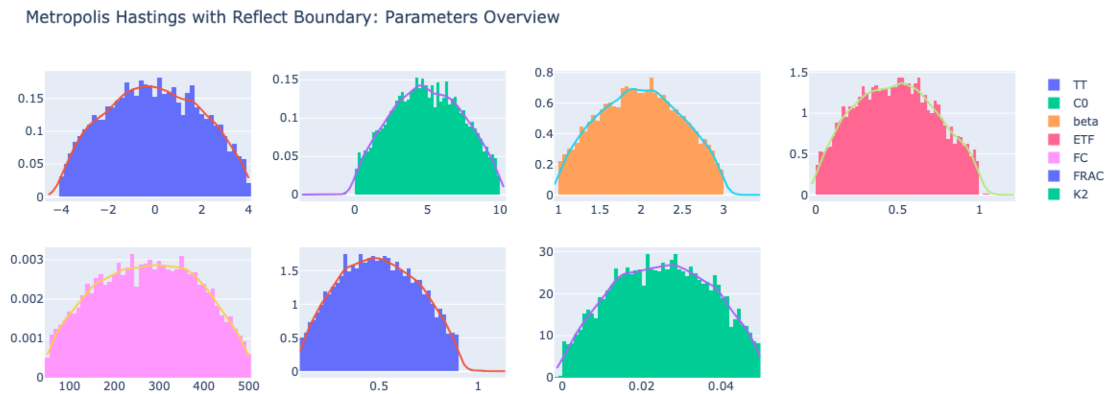


Figure 5.6.: Overview of the posterior distribution of the parameters calibrated by the Metropolis-Hastings algorithm that reflect the samples into the inside of the range if the they are out of bounds

The boxplots of the parameters are shown in Figure 5.7. Due to the normal distribution like posterior, all of the boxplots shrink by a certain amount, with some having lower upper bounds or upper lower bounds. This is an apparent result, since the shape of the normal distribution focuses on the mean, whereas the sampling probabilities of samples that are further from the mean are lower. This property can be presented by the boxplots.

Figure 5.7.: Boxplots of the parameters calibrated by the Metropolis-Hastings algorithm that reflect the samples into the inside of the range if the they are out of bounds

To find out which of these three variants delivers the best result, each of these three versions is separately executed, while all of the rest input algorithm parameters are set to the same value or instance. The different metrics that are mentioned in the section above are then calculated and visualized using a bar chart so that the values can be compared. The result is shown in Figure 5.8. The first impression of the bar chart is that the accuracy of the actual inferred results is pretty similar among all three versions, with the ignoring and the aggregate methods performing only slightly better. For efficiency, however, the ignoring performs better than both of the other methods by a huge margin. The reason behind it is obvious: the ignoring operation is way more efficient than the reflecting boundary and aggregation, which involves mathematical operations. Therefore, the default ignoring method is the clear winner here and should be retained for further model executions.

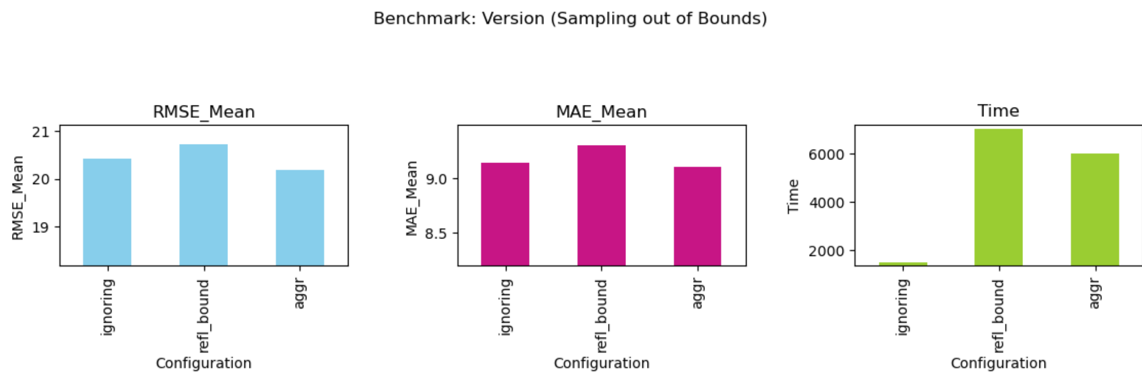


Figure 5.8.: Comparison of the accuracy and the efficiency of Metropolis-Hastings algorithms based on the technique of handling samples generated out of bounds

5.3.2. Sampling Kernel

The sampling kernel plays a crucial role in the accuracy and efficiency of the performance. Since we stick to the normal distribution in this thesis, the standard deviation is the only value that needs to be explored. As suggested in the above section of 5.2, the default value of the standard deviation is set to 6 because it is the largest possible number that fits in the use case of the hydrology model. However, the final result would also be different if the standard deviation is less than the optimal one. In this case, the movement of the samples is going to be relatively centered local, since the points in the vicinity of the mean are more likely to be sampled. To test multiple scenarios and their behaviors, three values for the standard deviation are going to be tested in the following execution: the range interval over 8, over 10, over 12, over 18, and over 24.

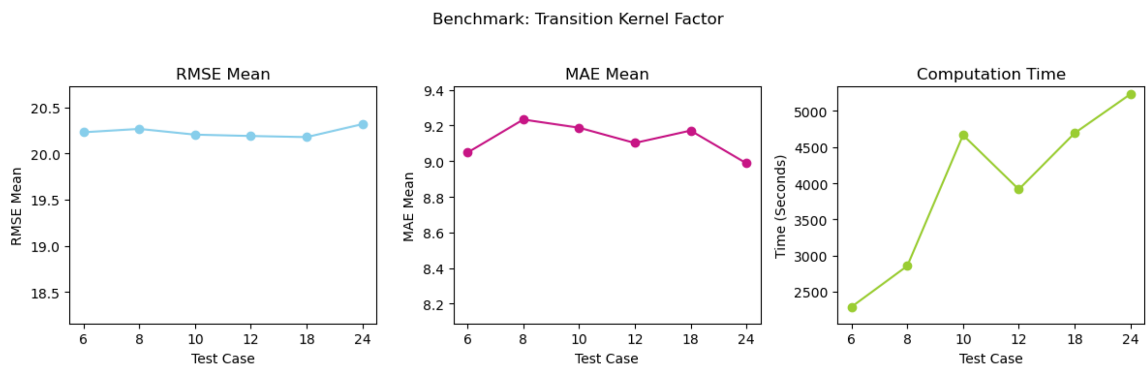


Figure 5.9.: Comparison of the accuracy and the efficiency of Metropolis-Hastings algorithms based on the sampling kernel standard deviation

The result shows a certain level of dependency between the accuracy metric and the input values. For both accuracy metrics, the accuracy scores the input value do not differ that much from each other, which indicates that the sampling kernel factor is irrelevant for the accuracy result. For the computation time, on the other hand, there is an exponential relationship between the run time and the sampling kernel value, with an exception of the test case 10 as anomaly. In conclusion, a lower value such as 6 for the sampling kernel factor would be optimal due to the efficiency of the algorithm.

5.3.3. Likelihood Functions

At the start of the chapter, we have already discussed the importance of the role played by standard deviations in the likelihood functions. As mentioned before, 1 is selected for the default value. However, the choice of the standard deviation does impact the final result, since it exerts an influence on the sampling probability of the values based on their distance from the mean. If the standard deviation is set lower, the sample that is away from the mean will receive less probability. If the standard deviation is set higher, that sample will not receive that little probability, which allows more tolerance to be present in the calculation. However, the value cannot be too large, otherwise, the tolerance level will be too high for the likelihood function to give out a meaningful solution. Thus, for the test values, we go from 1

down to 8, which is an interval of values that still might generate meaningful calculations. The selected values for testing are 1, 3, 5 and 8.

As we can derive from Figure 5.10, the metrics for the mean also do not differ that much from each other. In this case, the conclusion could be drawn that the standard deviation does not have a huge impact on the actual inferred result. For the inferred maximum time series, the discrepancy is also not obvious, even though the test case 5 shows the most instability throughout the posterior. Nevertheless, using the test case 5 results in the most efficient calculation, whereas using the test cases 1 and 2 will require slightly more computation time.

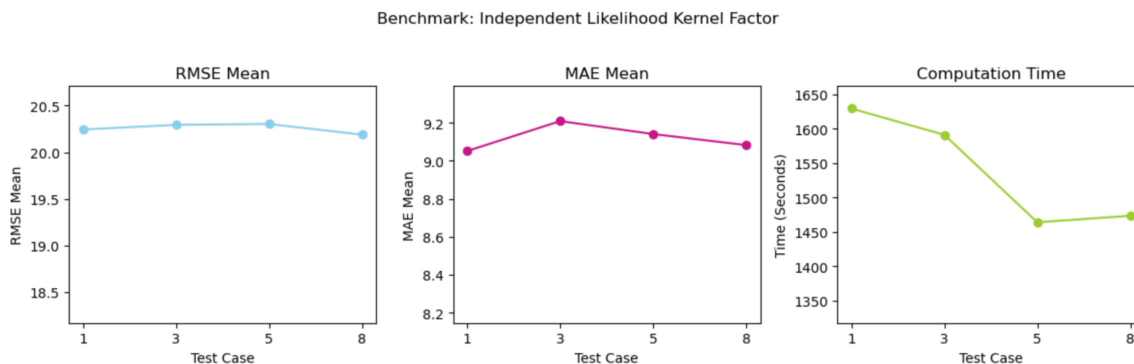


Figure 5.10.: Comparison of the accuracy and the efficiency of Metropolis-Hastings algorithms based on the likelihood function standard deviation

The above-computed likelihood function takes in a value as the standard deviation so that each sample is independently observed from another. Another alternative implementation to the dependent likelihood function would be the dependent likelihood function, which takes in the observed data as the standard deviation. This technique is common in the hydrological field [?], which uses the observed data as the standard deviation for a better understanding of the relationship between the inferred and observed data, therefore a precise likelihood calculation. Since the observed data might be too large for the likelihood function to deliver meaningful results, an alternative way is to take a certain factor of the observed data as input. Here, several factors are going to be tested, including 0.2, 0.4, 0.6 and 0.8. The goal is to observe the correlation between the accuracy and the value as it increases.

The result is shown in Figure 5.11, where we see a pattern: the accuracy of the inferred data decreases as the factor increases. As for the computation time, the case of 0.6 requires the most time to perform computation, and is, however, still pretty similar to the rest. Therefore, the case 0.2 should be the optimal choice.

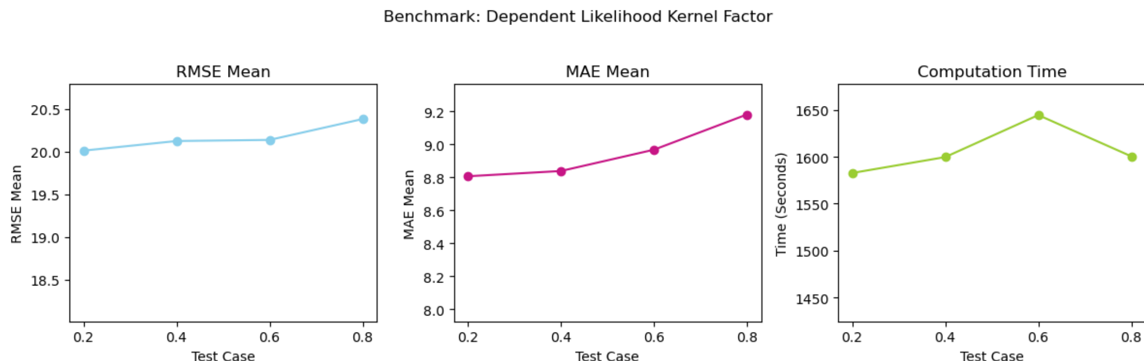


Figure 5.11.: Comparison of the accuracy and the efficiency of Metropolis-Hastings algorithms based on the dependent likelihood function standard deviation

Comparing the dependent and the independent version of the likelihood functions, the dependent version with a standard deviation of 0.2 times the measured data outperforms the independent version with a standard deviation of 5, despite a slightly longer run time, which is reasonable due to the mathematical computation of the standard deviation. The incorporation of exact data in the likelihood function does provide a more accurate result.

5.3.4. Alternative Implementation of Probability Acceptance Probability

After we explore both of the input algorithm parameters that are responsible for the calculation of the acceptance probability, we move on to the selection of the acceptance probability. As mentioned above, two choices are available for the acceptance probability calculation: One option is that we take the mean of all the values as an acceptance probability, so that the final acceptance probability could more generally represent all of the individual acceptance probabilities by parameter. Another option is to take the maximum value of the entire acceptance probability array. On the one hand, we can improve the efficiency of the algorithm, since the calculation of the mean is avoided. On the other hand, some dimensions might be easier to sample from than others due to less complexity. Using the maximum acceptance probability gives us therefore the insight of the entire parameter space, where the parameter with the best performance decides the acceptance probability. However, the maximum acceptance probability might be misleading if the distribution of the acceptance probability array is too widespread, which results in the complete opposite of efficiency and accuracy.

We execute the algorithm in both versions, with the rest of the parameters being identical. Figure 5.12 suggests that for the HBV-SASK model, the mean sampling method does not only deliver a slightly better performance in accuracy but also more stability and most importantly: better efficiency. This shows that the acceptance probability in each iteration in the array might have far different values so the max sampling method cannot deliver good enough results. Therefore, we stick to the original mean sampling method.

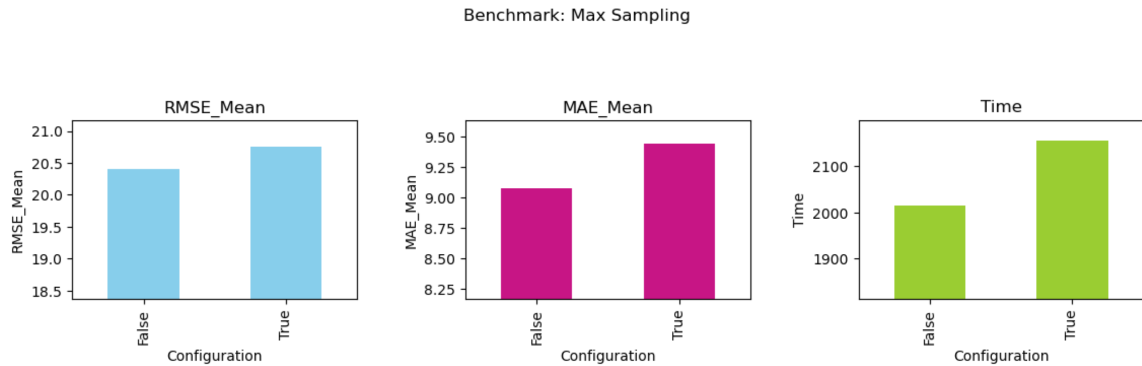


Figure 5.12.: Comparison of the accuracy and the efficiency of Metropolis-Hastings algorithms based on the implementation of probability acceptance probability calculation

5.3.5. Burn In Phase

Another crucial part of all Markov chain Monte Carlo algorithms is the discard of the burn-in phase. For Metropolis-Hastings, it is no exception. Determining a general optimal burn in phase leads to an optimization of the result since it is the process of removing generated samples that do not follow the stationary distribution and are unstable. The algorithm was previously executed with a burn-in phase of 20 percent, which means that the first twenty percent of the generated sample data are discarded. However, a comparison to other values of the burn-in phase would give us an overview of how effective the 20 percent is and whether it is enough. The values that are used for comparison are 33 and 50 percent. On the graph, the values 2, 3, and 5 denote the denominator of the burn in phase, as they should be interpreted as $\frac{1}{2}$, $\frac{1}{3}$ and $\frac{1}{5}$.

Figure 5.13 infers that the case 50 percent and the case 20 percent show relatively similar accuracy and efficiency. However, because removing half of the samples might be a bit too much and the similarity in performance, 20 percent is a better choice. Therefore, for the rest of the execution phase, we keep discarding the first 20 percent of the samples as the burn-in phase.

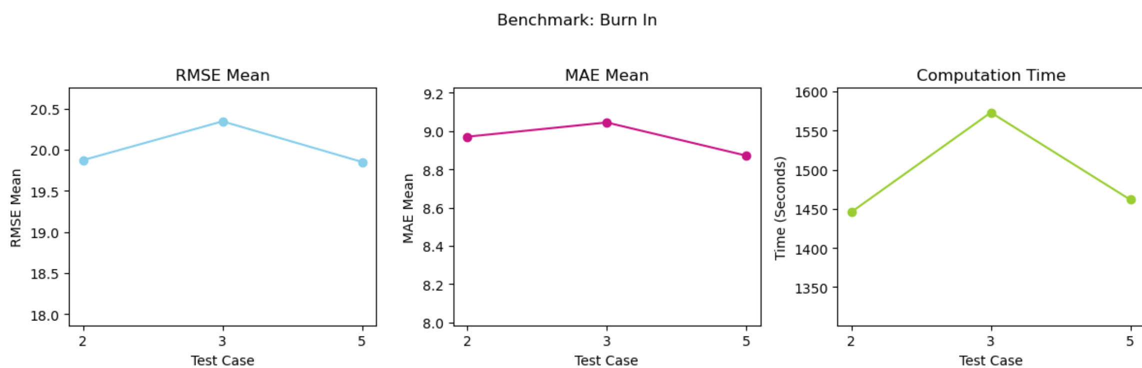


Figure 5.13.: Comparison of the accuracy and the efficiency of Metropolis-Hastings algorithms based on length of burn in phase

5.3.6. Effective Sampling Size

Effective sampling size is a new concept that has not been introduced in this thesis until here. It generally indicates the number of samples that are independent from each other. A basic way to implement an effective sampling size in the Markov chain Monte Carlo algorithms is to skip a few samples and only regard every several samples in our sample space. The reason to do this is that in Markov chain Monte Carlo algorithms, every sample is dependent on the last sample generated. However, this dependency might have a higher degree of influence, for instance: the newly generated sample is most likely to lie inside of a certain range depending on the standard deviation. Therefore, not considering the sample directly after another might lead to a certain level of independence, which gives rise to more generalization of the sampling space.

For testing purposes, we compare the algorithm that does not include the effective sampling size feature with algorithms that only consider every second, third, fourth, and fifth sample for the sampling space. The result is shown in Figure 5.14. The efficiency is proportional to the value set for the effective sample size, whereas no pattern could be found for the accuracy aspect. Even though only considering the third sample is relatively less efficient than considering every second sample or even not implementing this feature, it provides better accuracy than all the other cases and relatively better stability due to the low RMSE and MAE of the maximum time series. Therefore, it would be wise for us to keep retaining only the third value from the sample space in the later execution of the Metropolis-Hastings algorithm.

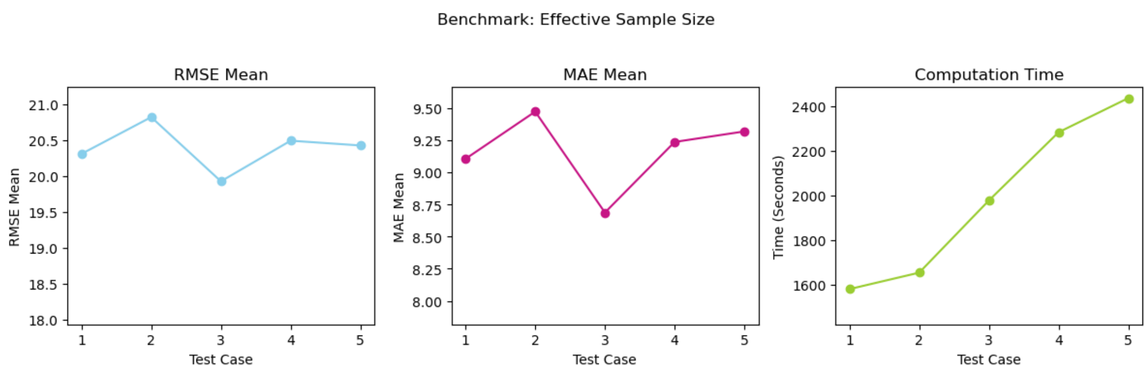


Figure 5.14.: Comparison of the accuracy and the efficiency of Metropolis-Hastings algorithms based on the selection of effective sample size

5.3.7. Iteration

Another crucial part of the Metropolis-Hastings algorithm is the amount of iterations. More iterations mean that more samples are gathered. To understand whether the amount of samples that are gathered is enough or not, comparing the accuracy with other numbers of iterations is needed. Until now, we performed the Metropolis-Hastings algorithm with 10000 iterations. Now we compare this number to other iterations like 5000, 20000, 40000, and 80000 to find out which number of iterations would deliver the best result and give a decent

efficiency.

The result is shown in Figure 5.15, where it is clear to observe that the computation time grows proportionally to the number of iterations. The case of 5000 delivers the best accuracy and efficiency but might lead to too small of a sample space due to the removal of the burn in period and effective sample size. Of all of the rest cases, they share a similar range of accuracy. However, due to the efficiency of run time, more than 10000 iterations will be an overkill. Therefore, we continue to execute the algorithm with 10000 iterations.

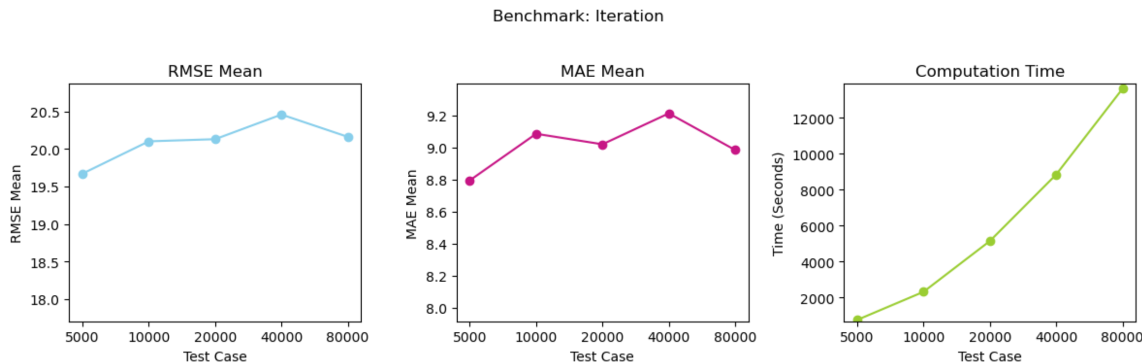


Figure 5.15.: Comparison of the accuracy and the efficiency of Metropolis-Hastings algorithms based on the number of iterations

5.3.8. Initial States

The last input algorithm parameter that needs to be explored is the initial states. The initial states should not have that much of an effect on the accuracy, but rather a great influence on the efficiency [?]. A better initial state would allow the algorithm to get rid of the burn-in phase and sample from the stationary distribution sooner, which reduces calculation burdens. It also allows the sampling kernel to discover more samples from the optimal ranges.

Several possible initial states should be taken into consideration. The most general one is the random initial state, which is used when there is little information regarding the distribution available. To do this, we sample a random state from the posterior and start from here. For testing purposes, however, we randomize 1000 samples and take the mean of them to maximize generalization and randomization. The lower and the upper boundary as the initial values would also work, which requires no initialization at all. Other possible values are derived from prior and posterior distributions. We try the first quantile, the mean, and the third quantile of the prior so that we can figure out whether an optimal starting value is coincidentally near these points. However, the focus point should be on the following three initial states: the first and the third quantile of the posterior distribution as well as the median of the posterior distribution. In the third section of this chapter, we generate a primary result that provides a general result of the posterior distribution. To start the entire algorithm from an inferred posterior state might result in better entry into the algorithm since the starting states are already proven to be very possible on the stationary distribution. Instead of taking the mean of the posterior, we select the median because it represents the half position of the entire posterior distribution, whereas the mean

only represents the middle value. It is expected that the most desirable solution comes from one of the proposals of the posterior.

We execute the model with all these different initial states and receive the results shown in Figure 5.16. The RMSE and MAE of the mean time series over all of the test cases prove that the performance is not influenced by the initial states. However, the efficiencies of the algorithms with different initial states do have a massive difference between them. The maximum initialization performs well, as well as the third quantile of the prior, the third quantile of the posterior, and the median of the posterior. After checking the numerical statistics, the median of the posterior delivers the most efficiency as expected. Therefore, it will be set as the default initial state.

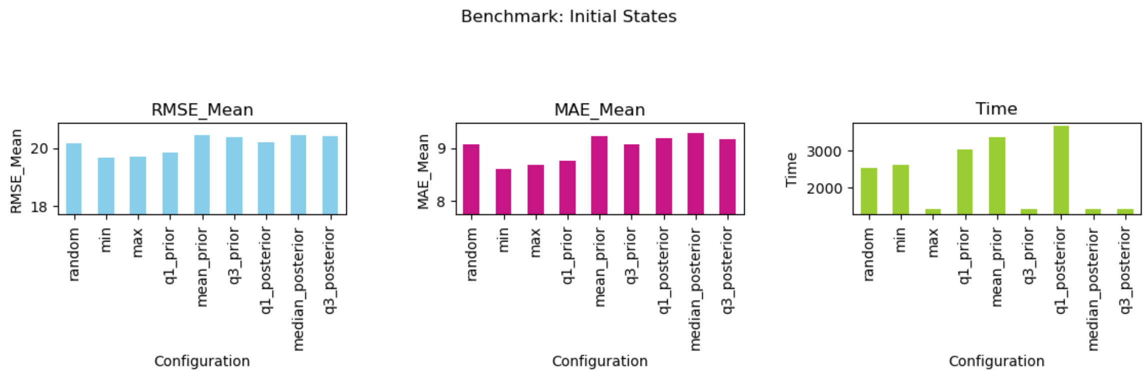


Figure 5.16.: Comparison of the accuracy and the efficiency of Metropolis-Hastings algorithms based on the selection of initial states

5.4. Result Comparison

After evaluating all of the input algorithm parameters, we will compare both sets of input algorithm parameters by using the Monte Carlo simulation. Using the set of input algorithm parameters that deliver the best performance by accuracy metrics in the exploration phase, which will, later on, be called the tuned input algorithm parameters, the model will be executed 1000 times, where the RMSE and the MAE of the result time series mean and maximum will be calculated. These results are going to be then compared to those of the models that run on the knowledge-based set of input algorithm parameters so that the set of input algorithm parameters that delivers better results will be selected to represent the basic Metropolis-Hastings method.

We first draw the histogram and the KDE plot for all of the parameters. It is shown in Figure 5.17. We can observe that the distributions fluctuate more than the parameters from the model that uses the knowledge-based input algorithm parameters. In this case, some of the parameters still show irregularity, whereas some of the distributions do show some certain level of resemblance to normal distribution, such as C0, FC, FRAC, and K2. Similar to the case before, the probability of sampling values near both boundaries is relatively low.

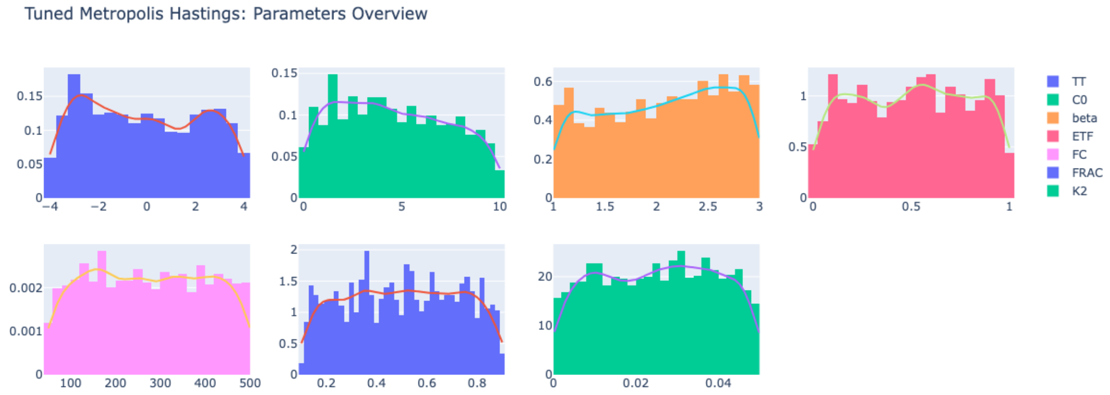


Figure 5.17.: Overview of the posterior distribution of the parameters calibrated by the Metropolis-Hastings algorithm with tuned input algorithm parameters

The boxplots of these parameters are shown in Figure 5.18. We can see that the ranges of most of the sampled parameters are different from the knowledge-based version. The beta and the C0 parameters have moved completely towards the lower bound, whereas the FRAC parameter has moved upwards. The ETF parameter retained its lower bound but had a lower upper bound. In contrast, The TT parameter retained its upper bound, however had a lower bound.

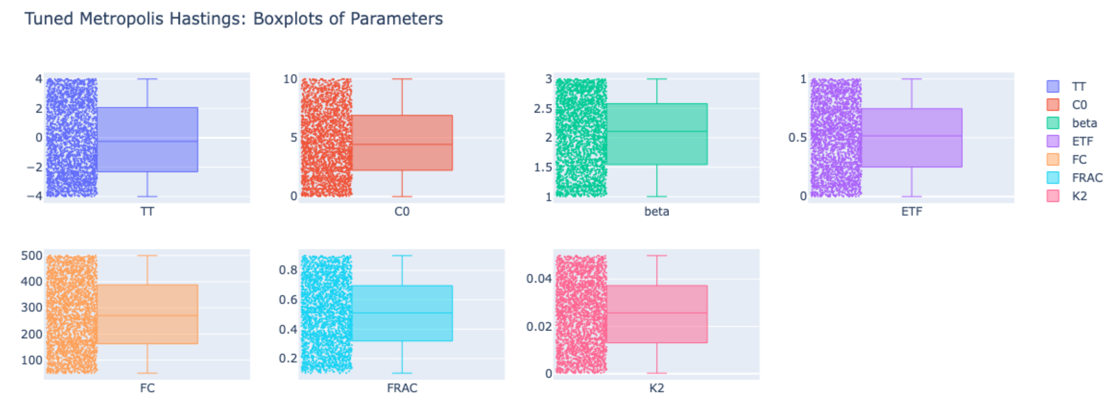


Figure 5.18.: Boxplots of the generated posterior samples of each parameter calibrated by the Metropolis-Hastings algorithm with tuned input algorithm parameters

From these generated samples, we retrieve the following result of the Bayesian inference problem shown in Figure 5.19.

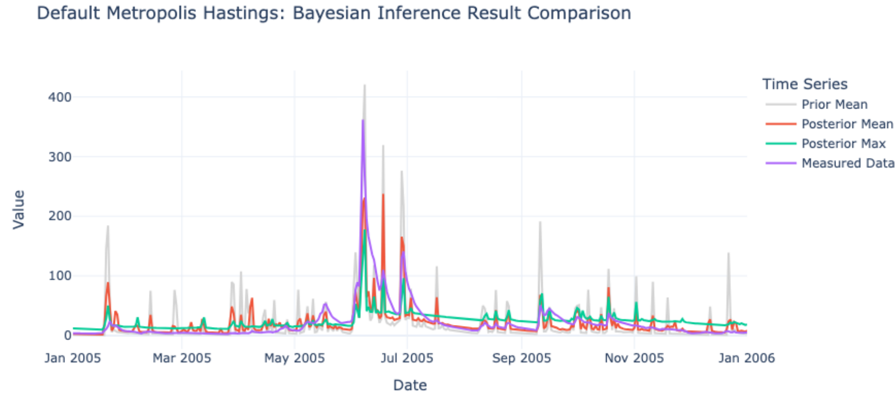


Figure 5.19.: Comparison of Bayesian inference results of the Metropolis-Hastings with tuned input algorithm parameters

To evaluate the result, we keep using the metrics that were calculated before for the model using the knowledged-based set of input algorithm parameters, namely RMSE and MAE. The same calculations are executed on the mean and the maximum of the inferred time series. The RMSE of the posterior mean is 21.974609013782757 and the MAE of the posterior mean is 11.457657543376751, whereas the RMSE of the posterior max is 24.458078992931487 and the MAE of the posterior max is 14.532783129590447. The model using the tuned input algorithm parameters performs better than the model using the knowledge-based input algorithm parameters in some metrics, but not the others. Besides, there is randomness in the Monte Carlo algorithm [**?**], which contributes to slightly different values in every single execution.

Therefore, to generalize the accuracy of the result, we run both models 100 times and gather the mean of all the metrics of the results.

Metric	**Knowledge-Based Posterior Mean**	**Tuned Posterior Mean**
RMSE	22.122504129857315	22.124942509212538
MAE	11.400067417022779	11.600318945622558

From this table, the knowledge-based input algorithm parameters achieve an all-around better performance, but not by much. While the RMSE does not differ from each other that much, the knowledge-based input algorithm parameters provide a slightly lower MAE. This might be the case that the inferred time series run by the model using the tuned input algorithm parameters performs slightly poorer in predicting extreme data points.

Therefore, we select the knowledge-based input algorithm parameters as the default input algorithm parameters for the Metropolis-Hastings algorithm for further usage in this thesis.

6. Parallel Metropolis-Hastings

In the chapter above, the fundamental Metropolis-Hastings were implemented and evaluated. In this chapter, we modify the fundamental Metropolis-Hastings algorithm, so that it can be run in a parallel way to optimize the run time.

6.1. General Idea of Parallel Metropolis-Hastings

In fundamental Metropolis-Hastings, we use a Markov chain to generate new samples. However, it would be possible to fully exploit the parallel computing property and run the algorithm with multiple Markov chains. The general idea would be to use multiple Markov chains instead of one single chain to generate samples. For instance, instead of drawing 10000 samples from one single Markov chain, we draw 2000 samples from five Markov chains simultaneously.

This modification leads to several aspects that should be discussed. For one, the combination of all of the samples could give rise to some problems. Since the Markov chain will eventually generate a sample from a given stationary distribution, all the samples generated across different chains will be sampled from the same stationary distribution. Therefore, we can simply combine the results after the burn-in period that are derived from different chains. Nevertheless, we need to make sure that all the samples that are generated from the chain after the burn-in period need to be sampled from the stationary distribution. Since each Markov chain from the parallel Metropolis-Hastings generates fewer samples than the amounts of samples that are generated by the Markov chain in the fundamental algorithm, it might be the case that some of the Markov chains do not reach the point where they sample from the stationary distribution. To examine this, we can use the trace plot [**?**]. It will be discussed in the section "Trace Plot" in this chapter.

Another problem would be the convergence of each dimension. After obtaining the final results, we need to make sure that each dimension of the Bayesian inferred result needs to deliver a converged result, which means that the entire result after the combination is stable and representative and reaches a stationary distribution. In a later section called "Gelman Rubin Convergence", we use the metric of Gelman Rubin Convergence to examine whether the Bayesian inferred result for the hydrological model using the parallel Metropolis-Hastings reaches convergence.

6.2. Efficiency Analysis

Since the machine, on which the algorithm is run, has 10 cores of CPUs, we are going to experiment with this algorithm with 10, 8, 5, 4, and 2 chains. These cases will be compared with the fundamental Metropolis-Hastings algorithm, where only one single chain is run. To analyze the algorithm concerning its efficiency, we record the run time of the algorithm using

different numbers of chains. The result is represented in Figure 6.1. It is obvious to see that the run time and the number of chains display an inverse proportional relationship. The more chains that are used, the less time is required for the algorithm to run, and vice versa.

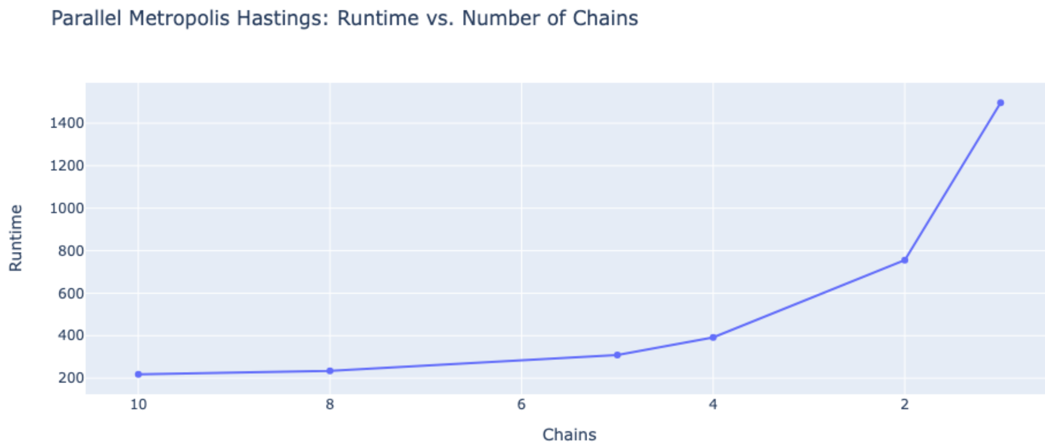


Figure 6.1.: Relationship between run time and chain numbers for parallel Metropolis-Hastings algorithm

6.3. Trace Plot

Making sure that each Markov chain generates samples from the stationary distribution is crucial in the parallel Metropolis-Hastings algorithm. The best way to examine this aspect is to visualize the trace of each sample generated from the chain [?]. The trace plot tracks each sample generated by the algorithm and plots the position of each sample by order. In this case, we can observe which sample is generated based on the last sample.

As mentioned before, it is more likely that the parallel Metropolis-Hastings algorithm with more chains shows less probability of sampling from stationary distribution than the parallel Metropolis-Hastings algorithm with fewer chains. This aspect is going to be analyzed now. We first select random chains from the algorithm run with the most number of chains, namely 10, and the least number of chains, which is 2, to observe the two extreme cases. Since 1000 samples are generated by each chain from the case of 10 chains, only 266 samples are going to be recorded after discarding 20% of the burn-in and using 3 as the effective sample size. With the case of 2 chains, the number of samples recorded has risen to 1333, which is almost 5 times as much as the previous case. This shows a higher probability that the samples are generated from the stationary distribution.

For both cases, two chains are selected for visualization to ensure the generality of the analysis. The figures of the trace plots are displayed in Figures 6.2 to 6.5. One major difference between the visualizations of both cases is that the chain for case 10 includes more abrupt jumps throughout the entire sampling process, whereas abrupt jumps occur less after the starting period is over for case 2. The longer chain in the case of 2 generally provides better stability and convergence of the parameter estimates. With no obvious

sampling trends and full exploration of the entire parameter space, it is observable that every parameter for the chain has settled into a stationary distribution that covers the entire parameter space. The shorter chain in the case of 10 may not enter the convergence and can be more susceptible to initial conditions or random fluctuations. Therefore, observing the trace plot for each chain after executing the algorithm is important, especially for algorithms that are run with more chains, each of which generates fewer samples.

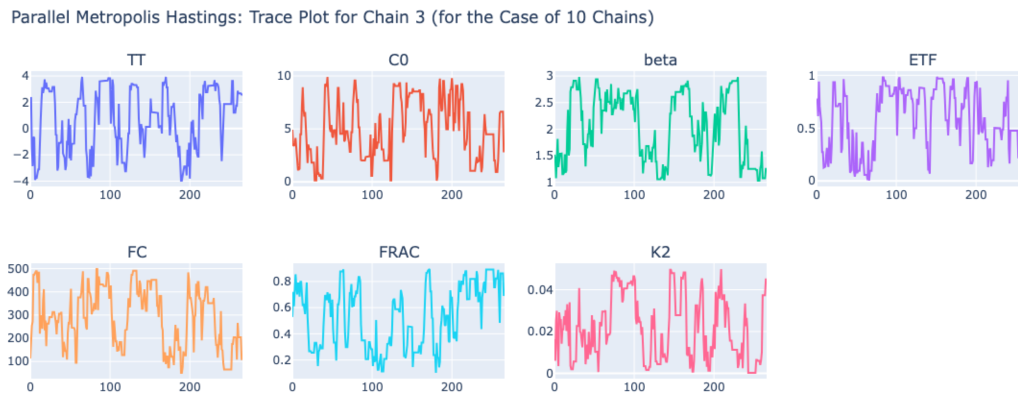


Figure 6.2.: Trace plot of the third chain from the parallel Metropolis-Hastings algorithm with 10 chains

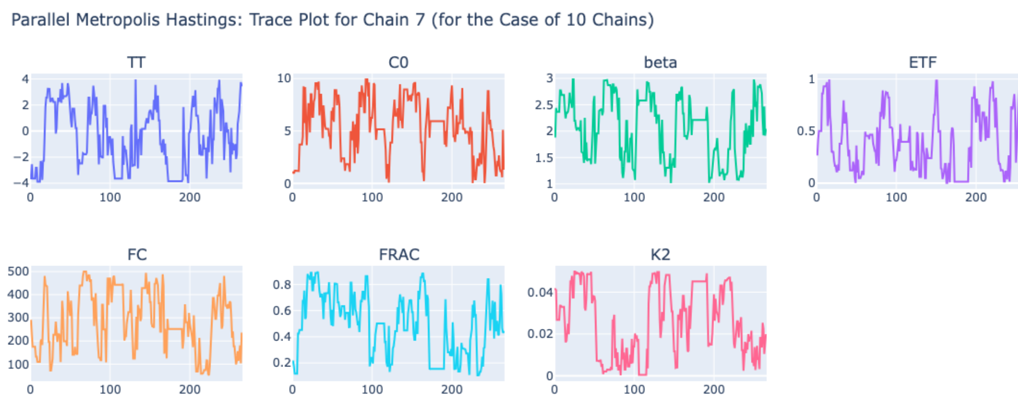


Figure 6.3.: Trace plot of the seventh chain from the parallel Metropolis-Hastings algorithm with 10 chains

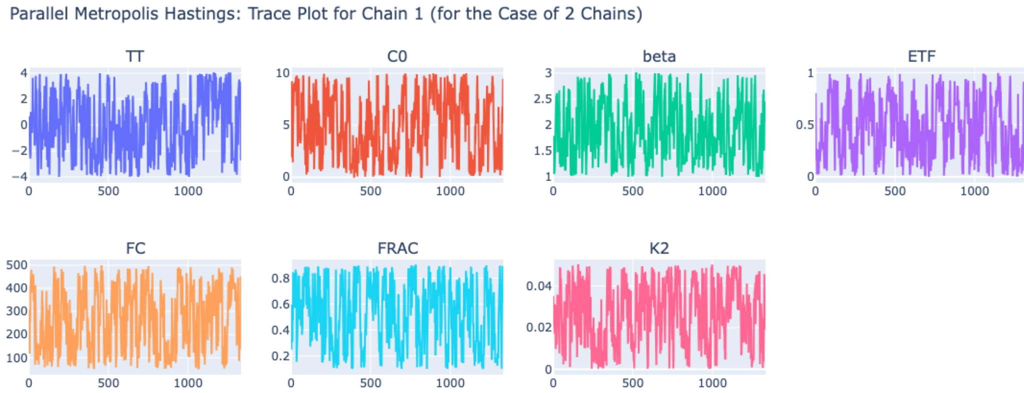


Figure 6.4.: Trace plot of the first chain from the parallel Metropolis-Hastings algorithm with 2 chains

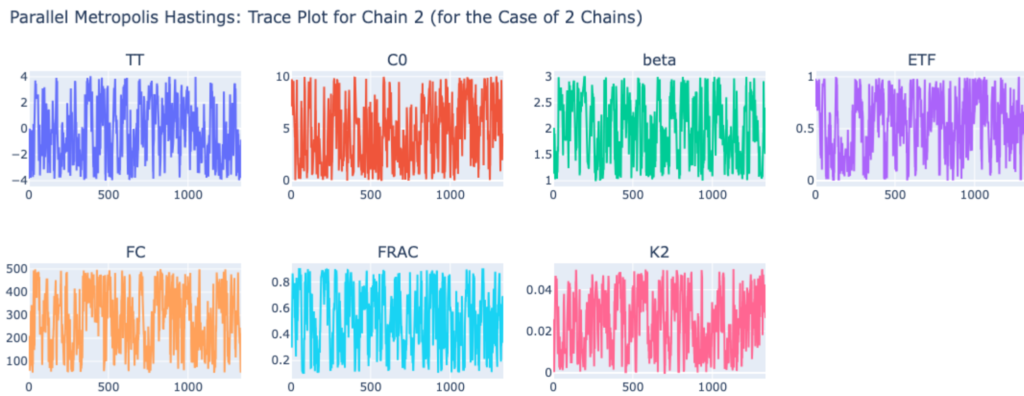


Figure 6.5.: Trace plot of the second chain from the parallel Metropolis-Hastings algorithm with 2 chains

6.4. Gelman Rubin Convergence

As we mentioned in the introduction of this chapter, the convergence of the result is an aspect that we shall observe after combining all results altogether. This diagnostic helps to determine whether the chains have reached a stationary distribution over time, which shows whether the samples are representative of the target distribution [?].

The calculation of Gelman Rubin's diagnostic looks as follows: We define m as the number of chains and n as the number of iterations of each chain. s_{ij} is then the vector of parameters in the ith iteration of from the jth chain. $\bar{s}_j$ is the mean of vectors within the jth chain, whereas $\bar{s}$ is the grand mean vector, calculating the mean of all of the $\bar{s}_j$ [?].

$$W = \frac{1}{m(n-1)} \sum_{j=1}^{m} \sum_{i=1}^{n} (s_{ij} - \overline{s}_j)^2 \tag{6.1}$$

$$B = \frac{n}{m-1} \sum_{j=1}^{m} (\overline{s}_j - \overline{s})^2 \tag{6.2}$$

$$V = \frac{n-1}{n} W + (1 + \frac{1}{m}) \frac{B}{n} \tag{6.3}$$

$$R = \frac{V}{W} \tag{6.4}$$

W is called the within-chain variance estimates, which estimate the variances of all sampled points within chains. B is called the average of the between, which measures the variance of the chain means around the overall mean of these means. V is the pooled variance estimate, which is calculated by the weighted versions of B and W. Last but not least, the ratio between the pooled an d within chain estimators is calculated and used as the Gelman-Rubin diagnostic. If the diagnostic is close to 1, commonly less than 1.1, it suggests that the chains have converged to the target distribution. A value greater than 1.1 indicates that additional sampling may be necessary, or that the chains have not yet mixed well and may potentially need more iterations [**?**]. For the case of MCMC, the threshold of 1.2 is also tolerated [**?**].

We now take a look at the performance of Gelman Rubin's convergence in the parallel Metropolis-Hastings algorithm. We take a look at the algorithms with different chain numbers, They are displayed in Figures 6.6 to 6.10. All of these cases show a good enough convergence to show compliance to the threshold of 1.1, even though a relationship between the convergence level and the number of chains could be found: the more chains there are, the less the result converges.

The reason behind this observation is that each chain generates fewer samples if more chains are used in the parallel Metropolis-Hastings algorithm. In this case, it is more likely that individual chains reach a low level of convergence since it does not generate enough samples. For instance, even though it still satisfies the convergence threshold, the case of 10 chains shows the worst convergence level of all of the test cases, since it only generates 1000 samples instead of 2500 in the case of 4 chains. Another observation is that no patterns can be found regarding the convergence behavior of individual dimensions. No parameter performs the best or the worst in every single case, which means that further investigation of individual parameters is not needed.

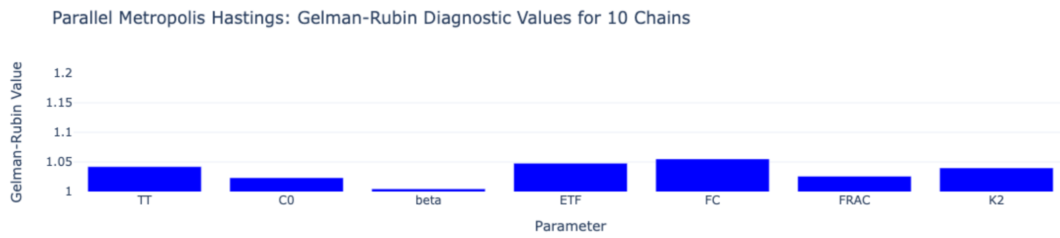


Figure 6.6.: Gelman Rubin Convergence Diagnostic of the parallel Metropolis-Hastings algorithm with 10 chains

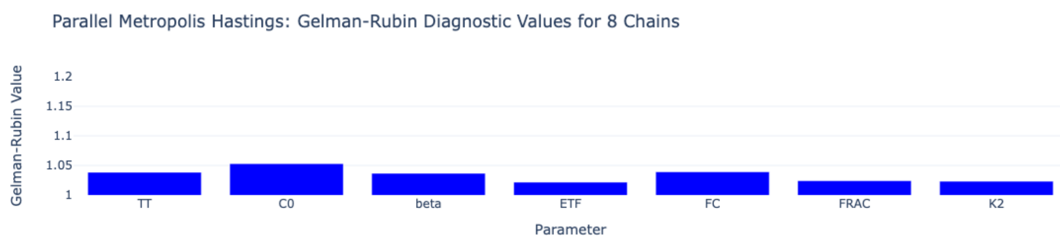


Figure 6.7.: Gelman Rubin Convergence Diagnostic of the parallel Metropolis-Hastings algorithm with 8 chains

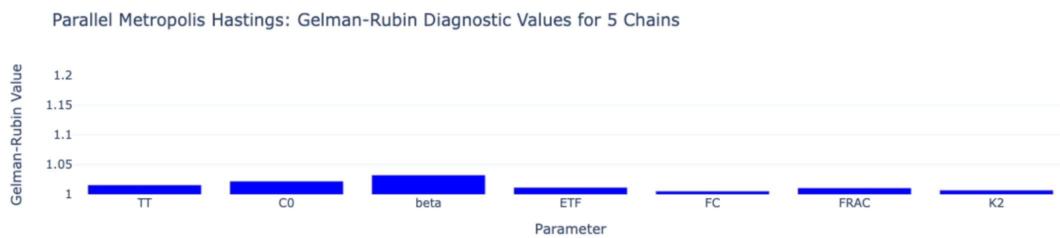


Figure 6.8.: Gelman Rubin Convergence Diagnostic of the parallel Metropolis-Hastings algorithm with 5 chains

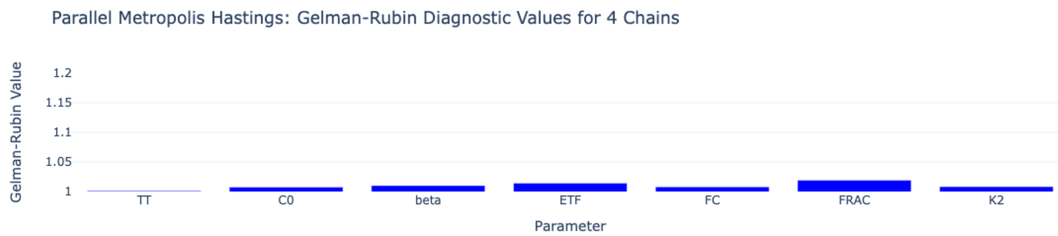


Figure 6.9.: Gelman Rubin Convergence Diagnostic of the parallel Metropolis-Hastings algorithm with 4 chains

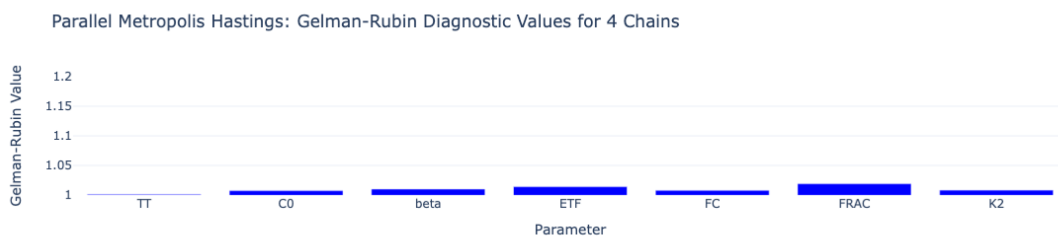


Figure 6.10.: Gelman Rubin Convergence Diagnostic of the parallel Metropolis-Hastings algorithm with 2 chains

6.5. Autocorrelation Plot

An autocorrelation plot displays the correlation of a series with itself at different levels of lags. In the context of MCMC, it shows the dependency of the current value in the chain and its past values. This plot is crucial because samples that are generated by Markov chain Monte Carlo algorithms are inherently sequential and may exhibit significant correlation with previous samples, which is something that should be investigated.

In this section, the autocorrelation of each of the Metropolis-Hastings cases regarding the number of chains is investigated. Since we draw the conclusion from the chapter above that the most optimal effective sample size is 3, we keep using this number in these test cases.

We first take a look at the cases with the most and the least number of chains, namely 10 and 2. These can be found in Figures 6.11 and 6.12. Both autocorrelation graphs share a characteristic that the autocorrelation drops to a low level quickly within a few lags. This suggests that the influence of any given sample on future samples diminishes quickly, which indicates good sampling efficiency. Afterwards, for both cases, the autocorrelation of all dimensions oscillates around 0 and 0.2 positive and negative, which suggests that the sampling shows stability and low correlation to past samples.

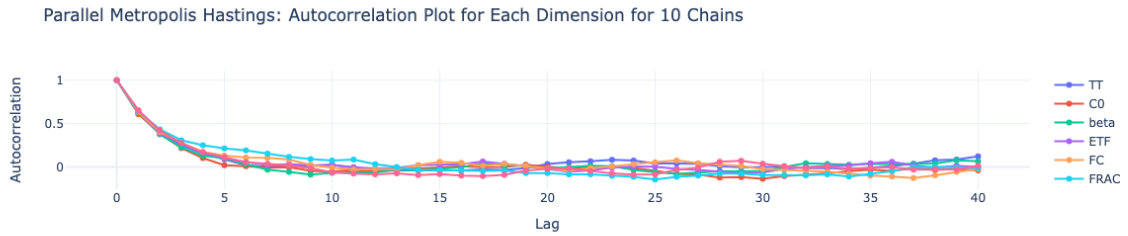


Figure 6.11.: Autocorrelation plot of the parallel Metropolis-Hastings algorithm with 10 chains

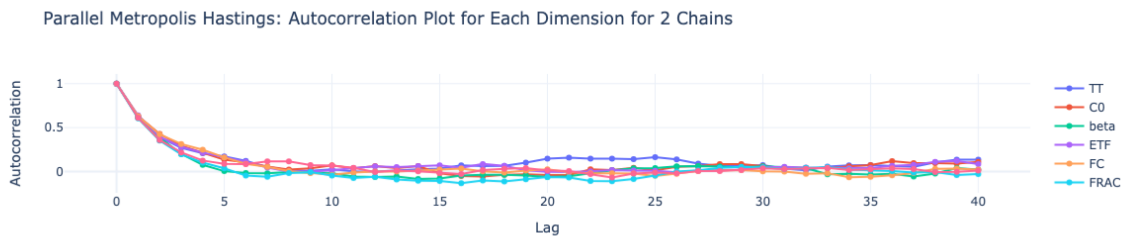


Figure 6.12.: Autocorrelation plot of the parallel Metropolis-Hastings algorithm with 2 chains

However, both graphs do not display an extremely low level of autocorrelation at high lags. For some parameters, the autocorrelation at high lags approaches 0.2, which is a relatively high value, even though it is generally acceptable. After plotting graphs for more test cases of the parallel Metropolis-Hastings algorithm using different chains, the autocorrelation graph of the parallel Metropolis-Hastings algorithm using 8 chains delivers a more optimal result. This can be seen in Figure 6.13. Autocorrelations around 20 lags approach closer to zero or show minimal fluctuation around zero. However, the autocorrelation around 40 lags shows a closer distance to 0, which is optimal for the independence property of Markov chain sampling. Nevertheless, all of the other cases of chain numbers show a satisfying result, with rapid diminishing in the first few lags, oscillation around zero, and a relatively low level of autocorrelation for all of the lags apart from the first few.

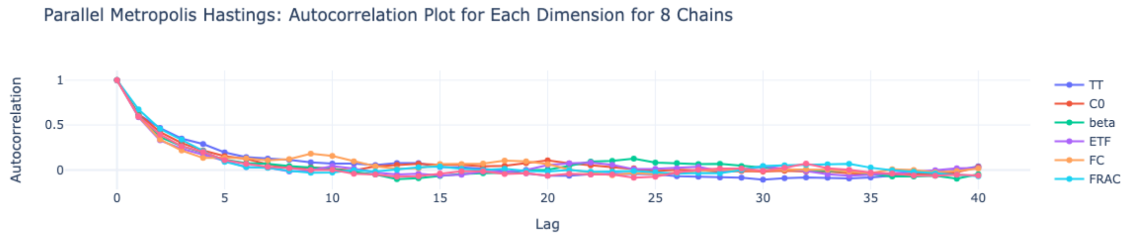


Figure 6.13.: Autocorrelation plot of the parallel Metropolis-Hastings algorithm with 8 chains

6.6. Accuracy Analysis by the Chains

After all these analyses regarding the components of the algorithm, we shall determine the number of chains that suit the algorithm the best. The accuracy metrics are kept the same as the ones that are used in the chapters above the RMSE and MAE. From the graphs shown in Figures 6.14 and 6.15, the parallel Metropolis-Hastings algorithm using 4 chains shows the best level of accuracy in both metrics. Centered around 4, the further the number of chains are, the less accurate they are, however not by a significant difference.

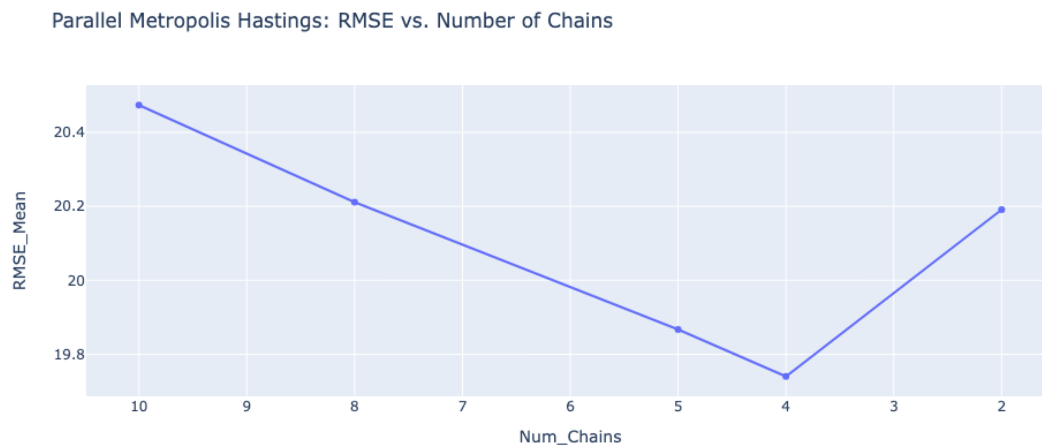


Figure 6.14.: Mean RMSE of the parallel Metropolis-Hastings algorithm across test cases with different chains

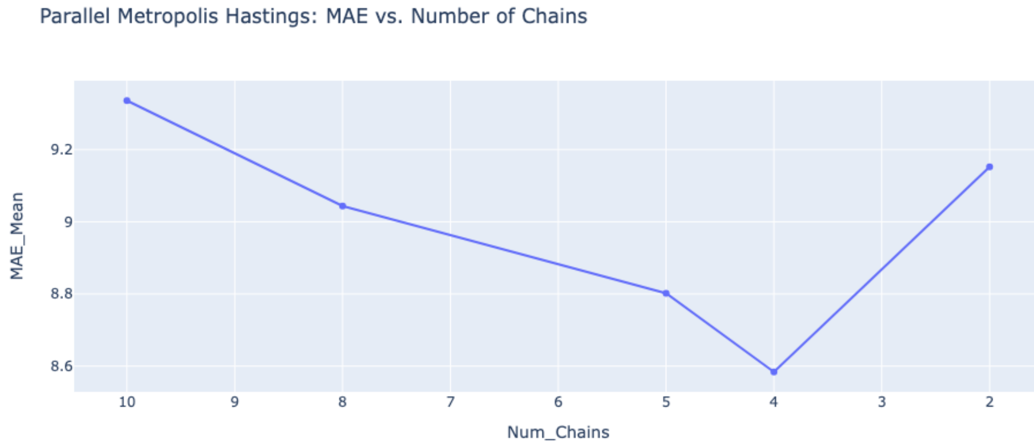


Figure 6.15.: Mean MAE of the parallel Metropolis-Hastings algorithm across test cases with different chains

6.7. Parameter Overview

The last aspect that is focused on in this chapter is the inferred parameter. In the chapter above, we visualized the inferred parameter by visualizing the distribution of the individual parameters. For the case of parallel Metropolis-Hastings, we utilize multiple chains to sample. Therefore, we shall not only just visualize the inferred parameter distribution of the entire result, but also visualize the inferred parameter distribution of each chain and to compare them with the result, to review the stability of the sampling of each chain and also finding out similarities and patterns.

As usual, we first analyze the parameters by a chain for the two extreme cases, namely parallel Metropolis-Hastings with 10 and with 2 chains. These can be found in Figures 6.16 and 6.18. From the case of 10 chains, we can see that the distribution of each chain is relatively random. To compose the distribution of the combined results, each chain is responsible for exploring a different region. For instance, for the parameter TT, the 9th chain explores the side with lower values more, whereas the 10th chain is more responsible for the side with higher values. Altogether, this property contributes to the relatively uniform distribution of the combined result. However, there are still similarities to some extent that can be found between the sample distribution of each chain and the sample distribution of the combined result. In each distribution, there are two peaks, each located on the left and the right side of a trough. This means that in each sampling process, two regions can be found that aggregate the most sampling values. Besides, both regions on the left and the right side of the interval acquire fewer samples than other regions, which contribute to the same property of the final combined result. In the case of 2 chains, things are not so complicated. Because each chain generated more samples than each chain does for the case of 10 chains, the distribution of each chain looks highly similar to the distribution of the combined result for most of the parameters. For the parameter of ETF, as an exception, it shares the property as the case for 10 chains, namely that both of the chains are responsible

for exploring two different areas, which are the left and the right sides of the entire interval.

Another plot that is used for visualizing the parameters is the boxplot, which is also visualized here in the same way as above: each chain is visualized individually, where they are compared to each other and the combined result boxplot afterward. The figures for cases of 10 chains and 2 chains can be found below in Figures 6.17 and 6.19. The results that we can draw from observing these boxplot graphs are the same as the conclusion that we have drawn above for the case of more chains, the 1st quantile, the median and the 3 quantile all vary from each other, which means that each chain is responsible for different regions. For the case of fewer chains, however, the 1st quantile, the median, and the 3 quantile lie almost on the same level or do not vary from each other that much, indicating that the samples from each chain are more stabilized.

Figure 6.16.: Parameter overview by chain for parallel Metropolis-Hastings using 10 chains

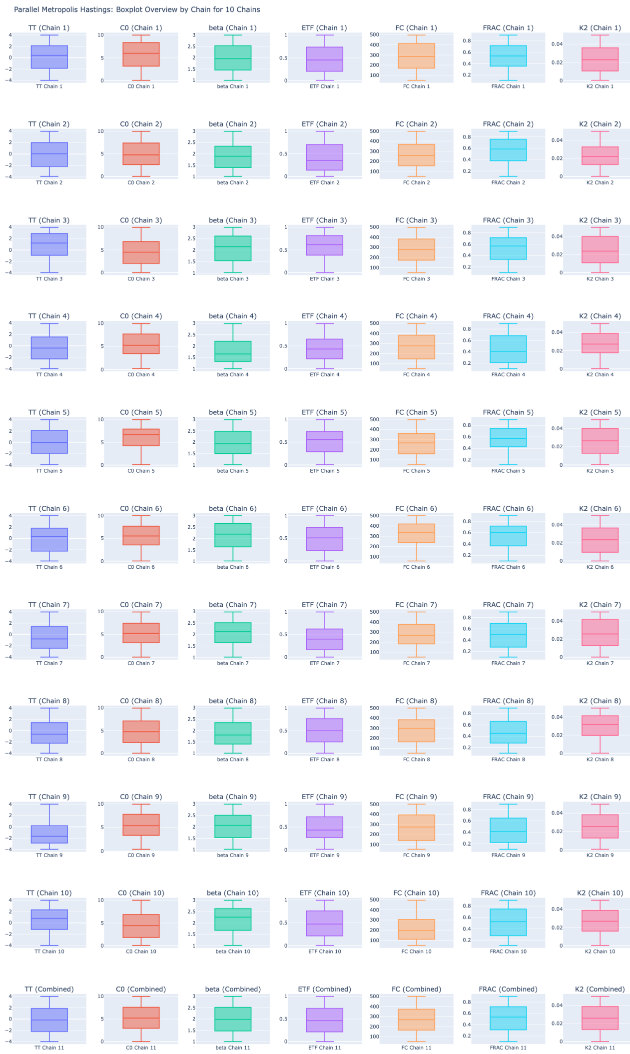


Figure 6.17.: Boxplot by chain for parallel Metropolis-Hastings using 10 chains

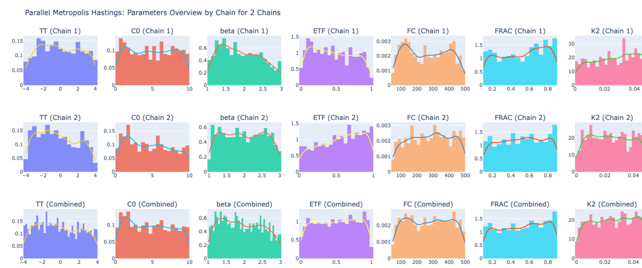


Figure 6.18.: Parameter overview by chain for parallel Metropolis-Hastings using 2 chains

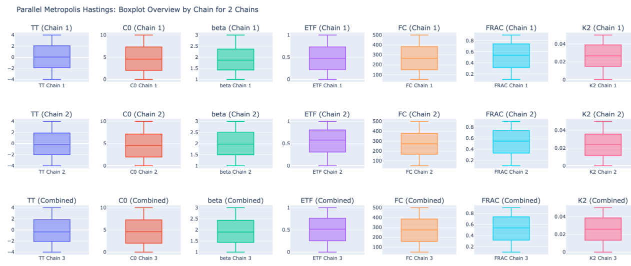


Figure 6.19.: Boxplot by chain for parallel Metropolis-Hastings using 2 chains

7. General Parallel Metropolis-Hastings

7.1. Introduction of the Algorithm

The general parallel Metropolis-Hastings algorithm is an algorithm that is based on the fundamental Metropolis-Hastings algorithm with some more parallel modifications. Instead of using different chains to make the algorithm run parallel, the general parallel Metropolis-Hastings algorithm applies a different idea: it generates multiple samples instead of one single sample in each iteration. This idea of parallel execution is inspired to avoid sequential generation of data [?], which means that instead of accepting or denying the generated sample, a different way to determine the acceptance of the generated sample points needs to be come up. Since the last sample before the generated sample is carried on if the newly generated sample is denied, we also take the last generated sample into account when designing the general parallel Metropolis-Hastings. However, since an array of points is generated, we will randomly select one of the samples as the starting point for the next round of generation [?]. Suppose that we draw m samples in each iteration, we then have to use $m+1$ samples to perform the acceptance or rejection. What we then do is to sample n points from these m samples randomly. We construct a probability space by calculating the likelihood for each sample point and sample a subset of these points as the points that should be added to the results.

A detailed explanation of the calculation of acceptance probability is inspired by the derivation from the GMH library of the MUQ framework.[1] The first step is to construct the acceptance matrix. This matrix has the dimension of the acceptance probability matrix A is calculated as follows:

$$A_{ij} = \begin{cases} \min\left(1, \frac{\exp(r_j - r_i)}{m+1}\right) & \text{if } i \neq j \text{ and proposed state } j \text{ is not None} \\ 1 - \sum_{k \neq i} A_{ik} & \text{if } i = j \end{cases} \tag{7.1}$$

where r_j is the log density of the jth proposed state.

We then calculate the stationary acceptance distribution. It is denoted as π, which is a vector with a length of $m+1$. To calculate this, we construct the matrix $M = A^T - I$, on which an extra row of 1 is added at the very bottom that represents the weight of the last generated sample. To calculate the stationary distribution, we create the b vector with 0 everywhere except for the last position, in which a 1 is set for the same reason mentioned above [?]. The calculation is then listed as follows:

$$M\pi = b \tag{7.2}$$

After acquiring the acceptance probability vector, we construct a probability space using it, in which we calculate the sum of the vector and then normalize it. Using this vector, we

[1] https://bitbucket.org/mituq/muq2/src/d99f6124bf142922d7973cc60c25d7a518084d12/modules/SamplingAlgorithms/src/GMHKernel.cpp?at=master#GMHKernel.cpp

generate n samples from it, including the sample that was generated before. We iterate this process for a certain amount of time and receive the result. To sum up the entire process, we use pseudo-code to illustrate it.

Algorithm 2: General Parallel Metropolis-Hastings Algorithm

Input: proposal distribution function, sampling kernel function, likelihood kernel function, initial state, number of iterations, acceptance_rate_calculation, num_proposals, num_accepted
Output: list of sampled data points

```
1  Function GPMH(proposal_dist, sampl_kernel, likel_kernel, init_state, iterations,
     acceptance_rate_calculation, num_proposals, num_accepted):
       // Initialize the samples list with the initial state
2      samples ← [init_state]
3      old ← init_state
4      for i ← 1 to iterations do
           // Generate a new sample from the sampling kernel
5          generated_samples ← [old]
6          for j ← 1 to num_proposals do
7              generated_samples.append(sampl_kernel(old))
           // Calculate the acceptance probability
8          acceptance_rates = acceptance_rate_calculation(generated_samples)
           // Decide to accept or reject the new sample
           // random_sampling is a function that takes two parameters,
              sampling randomly the number of times given in the first
              parameter from the array given in the second parameter using
              the acceptance probability provided in the third parameter
9          res ← random_sampling(num_accepted, num_proposed, acceptance_rates)
10         samples.append(res)
11         old = random(res)
12     return samples
```

7.2. Evaluation

After discussing the algorithm itself, we run the algorithm and generate data that can be used to analyze. In this section, we will first try to observe the algorithm regarding the number of samples generated and accepted in each iteration through ratio and amount tests. Afterward, further investigation regarding input algorithm parameters is conducted.

7.2.1. Ratio Test

The first test for evaluation is called the ratio test. In this test, we investigate the ratio between the numbers generated and the numbers accepted. This test is conducted for us to find the most optimal ratio of both parameters for the Bayesian inference problem. We fix the number of accepted samples being five, while the number of generated samples is a

variable. The tested scenarios include 5 generated samples for a ratio of 1, 10 generated samples for a ratio of 2, 20 generated samples for a ratio of 4, 40 generated samples for a ratio of 8, and 80 generated samples for a ratio of 10. As we can see from Figure 7.1, the accuracy of the Bayesian inference goes up as the ratio of the generated sample amount against the accepted sample amount goes up. The larger the ratio is, the more accurate the Bayesian inference would be. However, the trade-off would be the run time, as the run time grows. There is an obvious exponential relationship between the run time and the ratio. Therefore, we need to consider the trade-off between accuracy and efficiency while selecting an appropriate ratio.

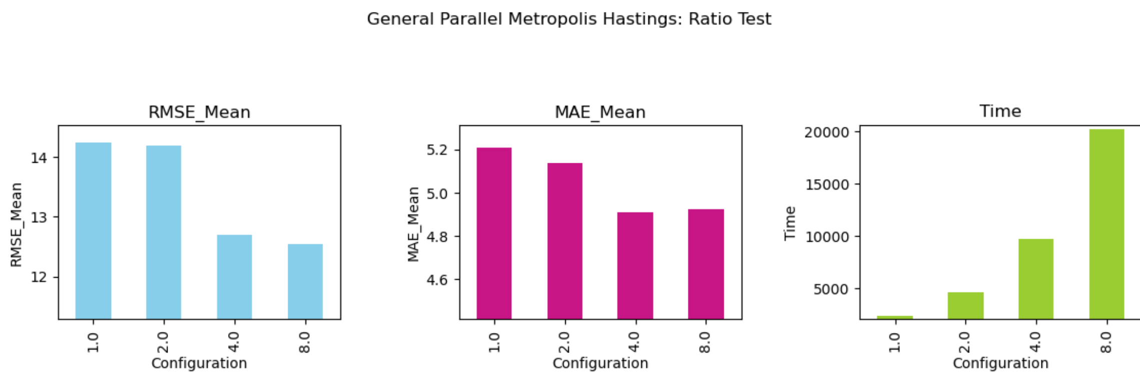


Figure 7.1.: Comparison of the accuracy and the efficiency of general parallel Metropolis-Hastings algorithms based on the ratio between numbers generated and accepted for each iteration

7.2.2. Amount Test

We move on to the amount test, in which we investigate the optimal amount of samples that need to be generated for each iteration. We fix the ratio between the generated samples and the accepted samples being 2. The tested scenarios include 10 generated samples with 5 being accepted, 20 generated samples with 10 being accepted, 40 generated samples with 20 being accepted, 80 generated samples with 40 being accepted, and 100 generated samples with 50 being accepted. In comparison with the ratio test, the differences are not that obvious in this case. For the RMSE, better performances of the Bayesian inference are delivered for higher amounts, whereas the MAE data do not differ too much from each other. For efficiency, faster execution happens also in higher ranges of amount numbers. Therefore, the more samples generated in each iteration, the faster and more accurate the algorithm will be.

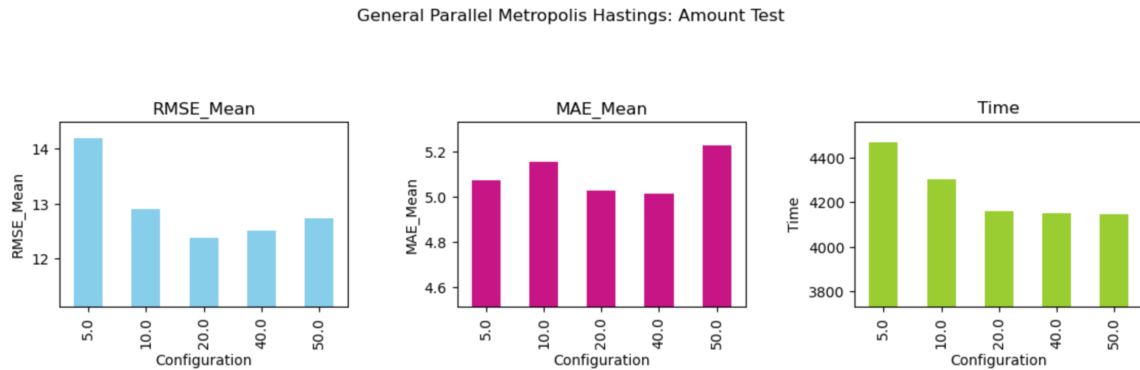


Figure 7.2.: Comparison of the accuracy and the efficiency of general parallel Metropolis-Hastings algorithms based on the amount of technique of handling samples generated out of bounds

7.2.3. Sampling out of bounds

After investigating the relationship between generated sample numbers and the accuracy and efficiency metrics, we now switch gears to the input algorithm parameters. For the first input algorithm parameter, we take a look at the sampling out-of-bounds methods. As it was mentioned before, we apply three methods when handling samples that are generated out of bounds: ignoring, reflecting boundary, and aggregation. These three methods are also used in the general parallel Metropolis-Hastings algorithm. The benchmark result is shown in Figure 7.3. Unlike the case in the fundamental implementation, all three variants show little difference from each other in terms of accuracy. In terms of efficiency, the ignoring method still outperforms the other two methods just like the case for the fundamental implementation, however not by a lot. The selection of the method usage is therefore not the most relevant selection for the general parallel Metropolis-Hastings algorithm considering the minimal impact on the accuracy and efficiency metrics.

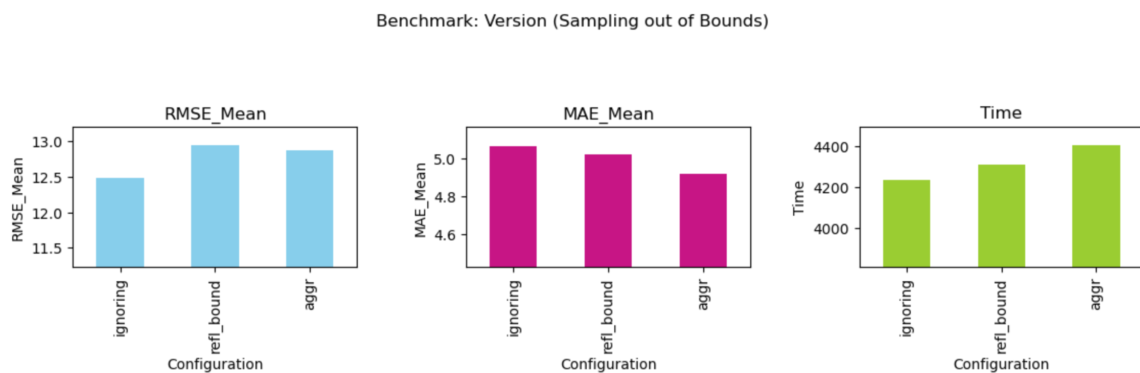


Figure 7.3.: Comparison of the accuracy and the efficiency of general parallel Metropolis-Hastings algorithms based on the dependent likelihood function standard deviation

7.2.4. Initial States

Another input algorithm parameter that might impact the resulting outcome is the initial states. The selection of the initial state does not result in a drastic difference in the accuracy metrics for the fundamental Metropolis-Hastings, and this is exactly the case here. For both metrics, the value for each input option of initial states does not vary much from each other. For efficiency, on the other hand, the initial state had a drastic influence on the run time. The fundamental algorithm with the best initial state could achieve less than twice the time that with the worst initial state. This is the complete opposite of the general parallel Metropolis-Hastings algorithm, in which algorithms with all different selections of initial states deliver similar results, all with a run time of around 4000 seconds. This means that the selection of the initial state does not have a big influence on the overall result of the general parallel Metropolis-Hastings algorithm.

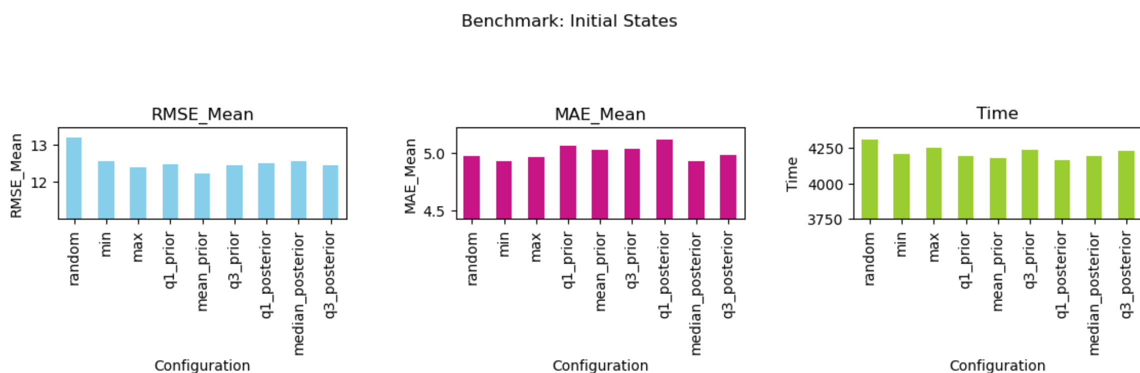


Figure 7.4.: Comparison of the accuracy and the efficiency of general parallel Metropolis-Hastings algorithms based on the selection of initial states

7.2.5. Dependent Likelihood Kernel Factor

The behavior of the dependent likelihood kernel factor for the general parallel Metropolis-Hastings algorithm is far different from the one for the fundamental Metropolis-Hastings algorithm. For the case here, the accuracy shows a certain level of irregularity, with the value 0.6 delivering the most optimal result. For efficiency, on the other hand, every run using different input values results in almost the same run time, except 0.8 as an anomaly, requiring more time than any other input values. Therefore, the value 0.6 is the most optimal choice for this case here.

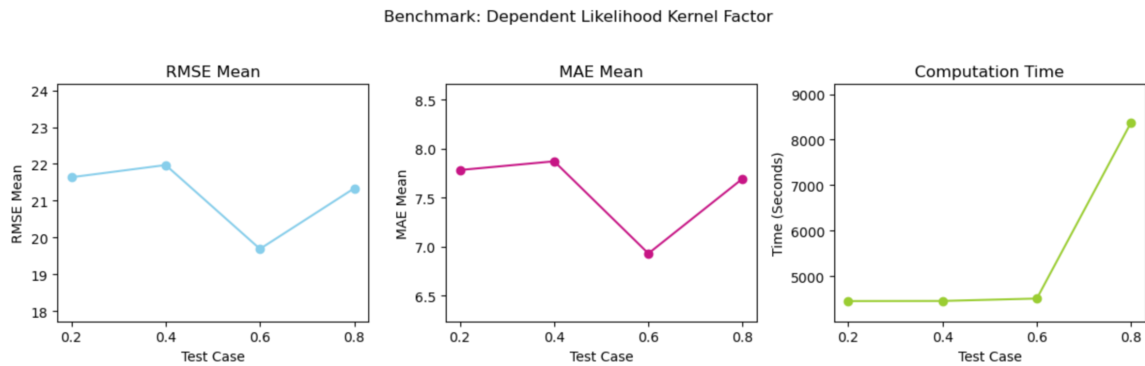


Figure 7.5.: Comparison of the accuracy and the efficiency of general parallel Metropolis-Hastings algorithms based on the dependent likelihood function standard deviation

7.2.6. The Rest of Input Algorithm Parameters

For the rest of the input algorithm parameters, detailed explanations are spared for this chapter, since they have similar results as the fundamental Metropolis-Hastings algorithm. These are listed here as follows:

- Transition kernel factor: Not many differences in terms of accuracy. For the efficiency part, the run time of the algorithm does not matter from each other too much, apart from an anomaly point at the very last, which has also occurred in the fundamental Metropolis-Hastings algorithm.
- Independent likelihood kernel factor: there is a peak on which the accuracy performance is the worst, 3 in the case of general parallel Metropolis-Hastings. Centered from this peak, the result gradually becomes more accurate. The run time for each run does not differ from each other by much.
- Dependent likelihood kernel factor: No big differences of accuracy across different input values. For the run time, both extreme input values (0.2, 0.8) provide the best performances, while the overall differences between the performances are not significant.
- Burn in: No big differences of both metrics across different input values.
- Effective Sample Size: No big differences of both metrics across different input values.

7.3. Parameter Overview and Comparison with the Fundamental Implementation

After performing a detailed analysis, the general parallel Metropolis-Hastings algorithm is run for one more time, so that the data output is gathered and used to be visualized. The Bayesian inference is run on the same dataset as one of the fundamental Metropolis-Hastings algorithms so that a direct comparison can be made.

The first graph gives an overview of the posterior distribution after the calibration. More information can be extracted from this posterior than one of the fundamental Metropolis-Hastings since there are regions for each parameter that gather more samples than others. Obvious peaks are presented so that the general shape of the posterior can be observed. The boxplots of all dimensions also display an aggregation of values from certain regions, making the parameters appear more concentrated and potentially indicating regions of higher posterior density. Therefore, we can conclude that the general parallel Metropolis-Hastings is a better choice than the fundamental Metropolis-Hastings in terms of the parameter calibration under uncertainty.

Figure 7.6.: Overview of the posterior distribution of the parameters calibrated by the general parallel Metropolis-Hastings algorithm

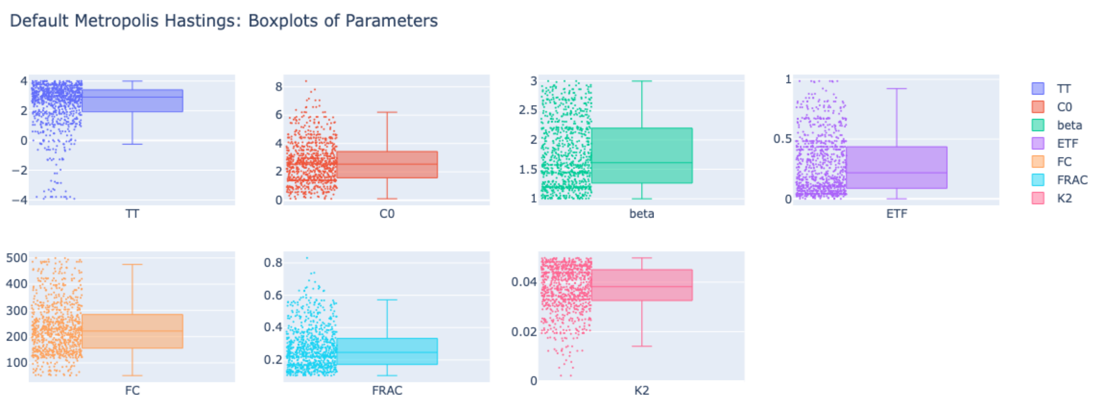


Figure 7.7.: Boxplots of the generated posterior samples of each parameter calibrated by the general parallel Metropolis-Hastings algorithm

In terms of accuracy, the RMSE score of the algorithm is around 19 and the MAE score is around 8.5 for most cases. This improves from the fundamental Metropolis-Hastings algorithm to a certain extent, which delivers an RMSE score of around 22 and an MAE score of around 11. However, the fastest run of the fundamental Metropolis-Hastings algorithm is around 1400 seconds in comparison to around 4000 seconds of the general parallel Metropolis-Hastings algorithm. Between better efficiency and better accuracy, there is a trade-off that needs to be considered.

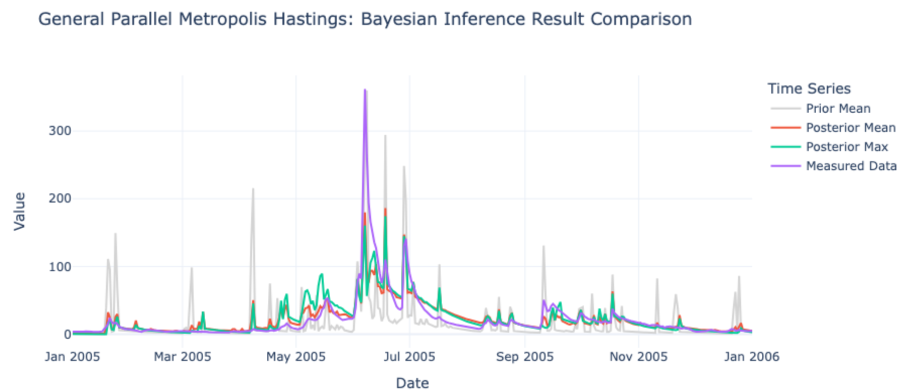


Figure 7.8.: Comparison of Bayesian inference results of the general parallel Metropolis-Hastings with tuned input algorithm parameters

8. The DREAM Algorithm

The DREAM algorithm is a Markov Chain Monte Carlo method that is designed for efficient sampling from complex and high-dimensional probability distributions. In this chapter, the algorithm is applied to the hydrological model. The output will then be analyzed and compared to the models from the chapters above.

8.1. Algorithm Introduction

The DREAM algorithm is designed based on the fundamental Metropolis-Hastings algorithm and the possibility of parallel execution of the algorithm. It applies differential evolution. It utilizes multiple Markov chains running in parallel to explore the parameter space. Proposals for new sample points are generated using the differences between pairs of chains. This differential evolution property helps to enhance sampling efficiency and convergence.

In the first section, we break down the DREAM algorithm step by step with details explained for individual components [**?**]. First, we take a look at the function header. Information regarding input algorithm parameter requirements should be provided.

1 function $[x, p_X] \leftarrow$ **DREAM**(prior, pdf, N, n, T, d)

The DREAM algorithm takes in several parameters as input. These include:

- *prior*: The prior distribution of the parameter space.
- *pdf*: The probability density function.
- N: Number of chains.
- n: Number of generations.
- T: Thinning factor, which is the implementation of effective sample size in the DREAM algorithm. Only the T-th sample will be retained in order to reduce autocorrelation [**?**].
- d: Dimension of the inferred parameter space.

The output would include x, which is the sampled chains positions, and their probabilities, which is denoted by p_X.

We move on to the next step, in which variables are initialized.

```
1 [delta, c, c_star, n_CR, p_g] ← [3, 0.1, 1e − 12, 3, 0.2]
2 x ← zeros(T, N, d)
3 p_X ← zeros(T, N)
4 [J, id] ← [zeros(1, N), 1 : N]
5 for i ← 1 N do
6   R(:, (i − 1) ∗ N + (1 : N)) ← setdiff(1 : N, i)
7 CR ← [1 : n_CR]/n_CR
8 pCR ← ones(1, n_CR)/n_CR
```

In this part, most variables that are internally used in this algorithm and not passed as the input algorithm parameters are initialized. These include:

- *delta*: Number chain pairs proposal. As mentioned in the introduction of this chapter, this value is used for observing the differential evolution of the sample space.
- *p_g*: The probability of selecting a specific number of pairs of chains for generating the proposals.
- c: The scaling factor that controls the step size of the proposal distribution.
- *c_star*: The parameter that is used to adaptively update the scaling factor c during sampling, so that an optimal acceptance probability can be controlled.
- J: The jump rate. It measures the average distance between successive samples in the parameter space and thus quantifies the effectiveness of the parameter space exploration.
- *id*: Chain indices that correspond the jump rate.
- CR: The crossover rate. It is used to determine the proportion of dimensions in which the state of the generated sample differs from the current state.
- *n_CR*: The number of different crossover values. It allows for adaptive updating of the crossover rates during sampling.
- pCR: The probability of each crossover rate being selected.

Next up, we initialize the sample states and probabilities.

```
1 X ← prior(X, d)
2 for i ← 1 N do
3   p_X(i, 1) ← pdf(X(i, :))
4 x(1, :, 1 : d) = reshape(X′, [1, N, d]); p_X(1, 1 : N) = p_X′
```

In this segment, the initial states are generated from the prior distribution and stored in X. The initial probability of each chain is also computed using the *pdf* function and stored in *p_X*. They are then reshaped correspondingly.

From now on, the algorithm enters the sample generation phase. It repeats itself until the completion of sample generation. All of the segments below are wrapped inside of a for loop that repeats until n cycles, which is the number of generation steps. The first segment inside of the loop looks like this:

```
1 draw ← sort(rand(N − 1, N))
2 dX = zeros(N, d)
3 lambda = unifrnd(−c, c, N, 1) std_X = std(X)
```

The **unifrnd** function is a function that generates random numbers from a uniform distribution. Here, we instantiated another few internal variables.

- *draw*: A list of random numbers. They are used to determine the order of chain updates.
- dX: An array to store proposal differences.
- *lambda*: A random variable sampled from a uniform distribution between $-c$ and c, used for adaptive scaling of the proposal step size.
- *std_X*: The standard deviation of X. It is used later for adaptive scaling.

Moving on to the next segment, we generate the proposals.

```
1 for i ← 1N do
2     D ← randsample([1 : delta], 1)
3     a = R(i, draw(D, 1))
4     b = R(i, draw(D + 1 : 2 ∗ D, 1))
5     d = randsample(1 : n_CR, 1, pCR)
6     z = rand(1, d) A = find(z <= CR(d)) if len(A) == 0 then
7         A ← min(z, c_star = 1)
8     gamma_d = 2.38/sqrt(2 ∗ len(A) ∗ p_g ∗ (1 − p_g))
```

The algorithm loops through every single chain and draws proposals.

- D: Selects a number of differences for the proposal generation.
- a, b: Chain indices. They are used in the differential evolution proposal.
- d: Selects a crossover rate using probabilities pCR.
- A: Determines the dimensions that are involved in the crossover. If the dimension turns out to be 0, we enforce 1 to be the minimum dimension size.
- *gamma_d*: The scaling factor for the differential evolution proposal. The calculation is based on the number of dimensions and a probability factor *p_g*.

For the next step, the algorithm focuses on the differential evolution proposal. This is one of the most crucial steps of the DREAM algorithm, making it different from other Markov chain Monte Carlo algorithms.

```
1 g ← randsample([gamma_d, 1], 1, [1 − p_g, p_g])
2 dX(i, A) = c_star + randn(1, len(A)) + (1 + lambda(i)) ∗ g ∗ sum(X(a, A) − X(b, A))
3 Xp(i, 1 : d) ← X(i, 1 : d) + dX(i, 1 : d)
4 p_Xp(i, 1) ← pdf(Xp(i, :))
5 p_acc ← min(1, p_Xp(i, 1)/p_X(i, 1))
```

In this segment, a few calculations is done to decide whether to accept the proposed move. This is done by generating a uniformly distributed random number and accepting the

proposed state if this number is less than or equal to the calculated acceptance probability, namely p_{acc}.

- g: Selects a number from *gamma_d* or 1.
- dX: The actual differential evolution proposal, calculated based on the dimension count of A, the variability term *lambda* that is defined above, and the g selected in the row above.
- Xp: The newly proposed positions.
- p_Xp: Density of the newly proposed positions.
- p_acc: The acceptance probability, calculated based on the newly proposed positions and their corresponding probabilities using the Metropolis criterion as the fundamental Metropolis-Hastings algorithm.

Afterward, we accept or reject the samples that are generated, the same as any other Markov chain Monte Carlo algorithm.

```
1 if rand < p_acc then
2 |  x(i,1,:) ← Xp(i,:) p_X(i,1) ← p_Xp(i,1)
3 else
4 |  dX(i,1:d) ← 0
5 J(i) ← J(i)sum((dX(i,1:d)./std_X).^2) id(i) ← id(i) + 1
```

The process of this step is relatively straightforward. A random number is drawn, so that the algorithm can decide if the proposal is accepted. If the proposal is accepted, the position and density of the chain are updated. Otherwise, dX is reset to zero for that chain. Afterwards, the jump rate J and chain index id are updated.

Before ending the repetition and continuing with the next step, there is an extra step in the DREAM algorithm which involves chaining and mixing.

```
1 x(:,:,:) ← reshape(X(:,1:d),[],N,d)
2 p_X(:,1:N) ← p_X.transpose
3 if t < T/10 then
4 |  pCR ← 1./J pCR ← pCR/sum(pCR)
5 [x,p_X] = check(X,mean(log(p_X(ceil(T/2):T,:))))
```

What the algorithm does in this part is to reshape and update the density of the chain's position for the next iteration. The crossover probabilities pCR are then updated based on the jump rate J to enhance mixing. At the very end, a check function is used to perform outlier detection based on the log probabilities of the chains, removing them and leaving the valuable data inside of the variable.

The loop is then ended. The final step of the algorithm is self-explanatory, namely the return phase. It returns the sampled values in the form of chains and ends itself.

```
1 return x,p_X
```

The algorithm itself is originally implemented using MATLAB, though it is later rewritten and offered in multiple packages with different implementation variants. For this thesis, the PyDREAM library[1] is selected for use. The PyDREAM library brings the DREAM algorithm to the platform of Python with easy installation and usage [**?**], which is optimal for the use case of this thesis.

8.2. Evaluation Based on Chains

For the DREAM algorithm, there is a set of default input algorithm parameters. Therefore, before exploring the influence of the input algorithm parameters on the actual output, we first use this set of default input algorithm parameters to run the algorithm to observe how the output of the Bayesian inference looks. Since it is an algorithm based on chains, we analyze the sampled results both with and without regard to chains, analogous to the parallel Metropolis-Hastings algorithm. For the test cases, we test the algorithm using 10, 8, 5, and 4 chains. Numbers of chains lower than 4 are not possible due to the policy of the DREAM algorithm, stating that the chain amount must be greater than twice the value of DEpairs + 1. The DEpairs parameter describes the pair of differential evolution and ensures there are enough chains to form the required number of DEpairs for proposal generation.

8.2.1. Efficiency

First, we take a look at the efficiency of the algorithm. We measure the run time for different test cases regarding chain amounts and compare them using graphics, which are displayed below in Figure 8.1. Unlike the parallel Metropolis-Hastings algorithm, there is no strict correlation that can be found between the number of chains and the run time of the DREAM algorithm. The cause is that instead of treating different chains individually, the DREAM algorithm uses cross-over calculations to track the relationships between the different chains. Therefore, the additional calculation results in the irregularity of the efficiency across different test cases of chain amounts.

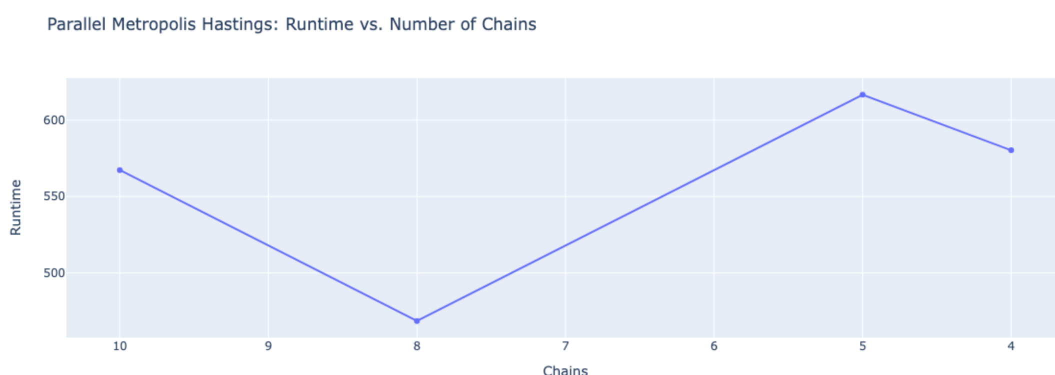


Figure 8.1.: Relationship between run time and chain numbers for the DREAM algorithm

[1] https://pydream.readthedocs.io/en/latest/index.html

8.2.2. Trace Plot

Similar to the case for the parallel Metropolis-Hastings algorithm evaluation, we use the trace plot to track the positioning of generated samples in each step. For the DREAM algorithm, we also analyze the trace plot of both extreme cases, namely the DREAM algorithm run with 10 chains and 4 chains. The trace plots of two random chains picked from both cases are shown in Figures 8.2 to 8.5. Visualizations for both cases do not differ much from each other in terms of sampling from the stationary distribution. All parameters in both graphs show apparent convergence approaching the end, where the stationary distribution can be easily observed. We could conclude that the property of sampling from the stationary distribution of individual chains exists and does not need further investigation.

Unlike the observation made for the parallel Metropolis-Hastings algorithm, the DREAM algorithm displays a heavy stationary distribution, where there are far fewer movements, more rejections during the sampling process, and no obvious moving patterns of the traces. Also, the visualizations show that the parameter space of the stationary distribution from the DREAM algorithm is normally a subset of the entire parameter space, which provides a more stable and consistent sampling, unlike the wide parameter sample space exploration upon stationary distribution in the case of parallel Metropolis-Hastings. In conclusion, the DREAM algorithm has a stronger convergence for the sampling process than other Markov chain Monte Carlo algorithms.

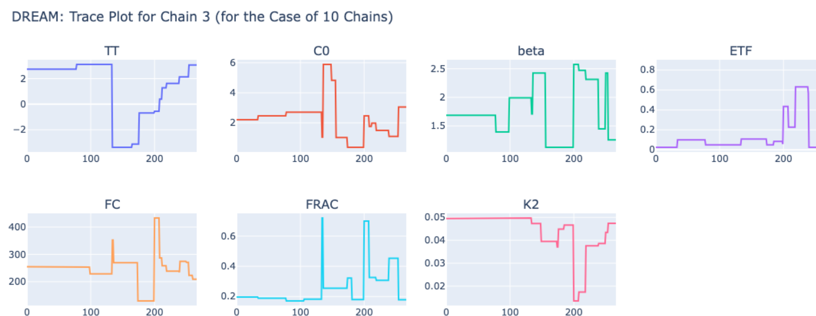


Figure 8.2.: Trace plot of the third chain from the DREAM algorithm with 10 chains

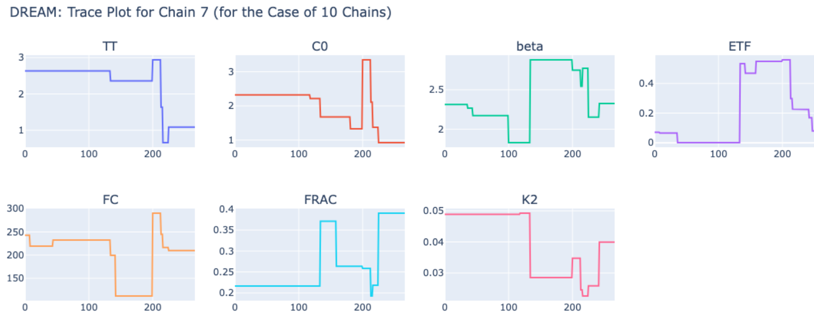


Figure 8.3.: Trace plot of the seventh chain from the DREAM algorithm with 10 chains

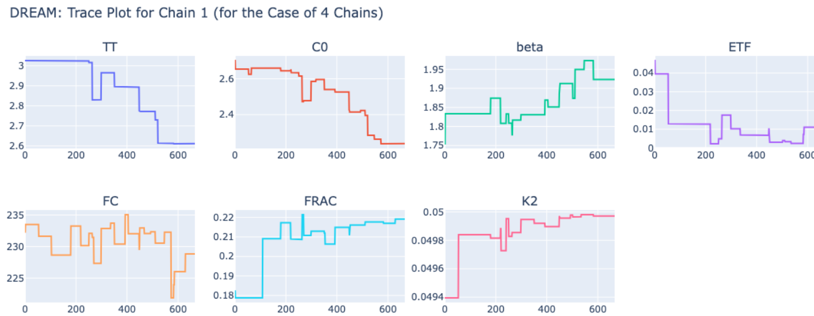


Figure 8.4.: Trace plot of the first chain from the DREAM algorithm with 4 chains

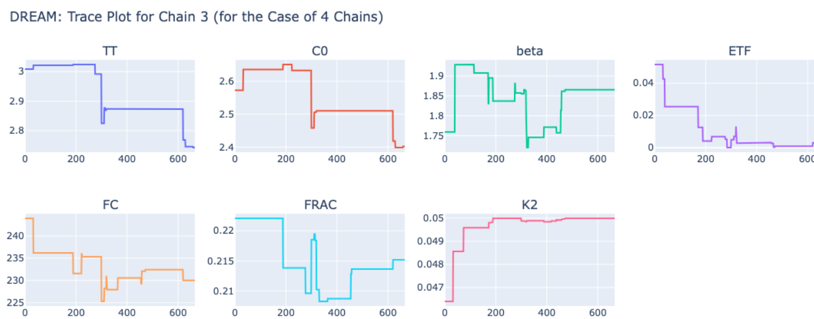


Figure 8.5.: Trace plot of the third chain from the DREAM algorithm with 4 chains

8.2.3. Gelman Rubin Convergence

The Gelman Rubin statistic in the case of DREAM is generally higher than the convergence diagnostic of the general parallel Metropolis-Hastings algorithm, even though the convergence statistic is generally in the acceptable range below 1.2. The figures for all test cases are displayed in Figures 8.6 to 8.9. For the case of 10 chains, two parameters show relatively high convergence diagnostic values that almost exceed the threshold. For other cases, there are also some parameters that show higher convergence diagnostic values than others. However, no specific patterns or regularities can be found for the Gelman-Rubin convergence parameter.

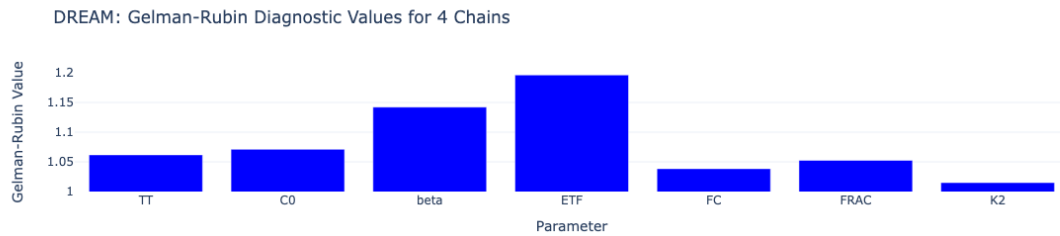


Figure 8.6.: Gelman Rubin Convergence Diagnostic of the DREAM algorithm with 10 chains

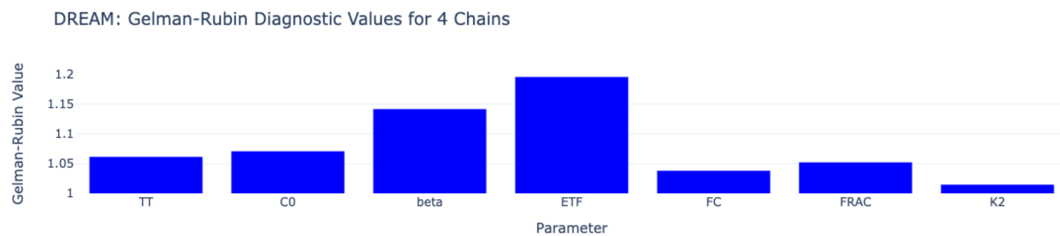


Figure 8.7.: Gelman Rubin Convergence Diagnostic of the DREAM algorithm with 8 chains

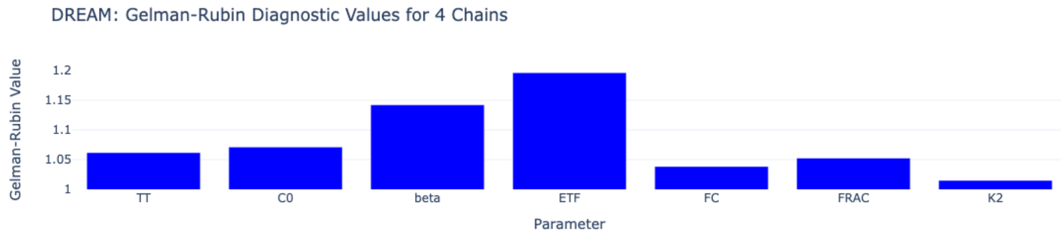


Figure 8.8.: Gelman Rubin Convergence Diagnostic of the DREAM algorithm with 5 chains

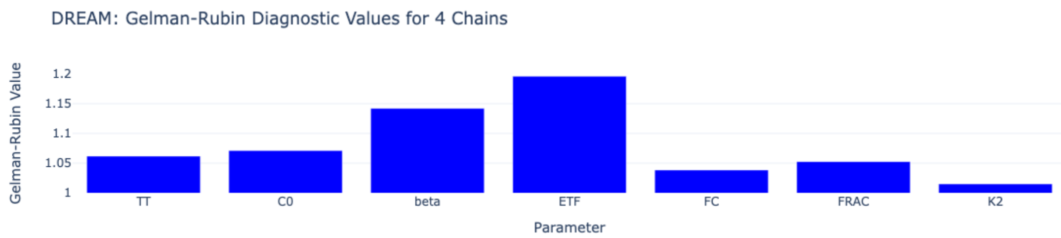


Figure 8.9.: Gelman Rubin Convergence Diagnostic of the DREAM algorithm with 4 chains

8.2.4. Autocorrelation Plot

We observe the autocorrelation plot for the DREAM algorithm to analyze the degree of sampling independence. At first sight, we can detect drastic different behaviors among all four cases. For the case of 10 chains in Figure 8.10, the descending of certain parameters is faster than some others. However, the TT and the K2 parameters show a high level of negative correlation, which means that the newly generated samples have potentially an inverse relationship to the samples generated before, indicating the lack of randomness in the sampling process. For the case of 5 in Figure 8.12 and 4 chains in Figure 8.13, the final autocorrelation for higher latency is generally in a favorable range. However, most parameters don't display a rapid descending, which might lead to inefficient sampling and longer convergence. The case of 8 chains displays the best autocorrelation plot among all four, including fast decrement and low level of autocorrelation throughout the entire range of latency.

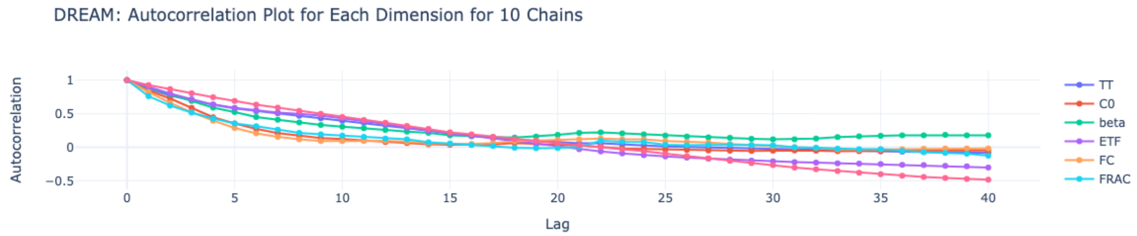


Figure 8.10.: Autocorrelation plot of the DREAM algorithm with 10 chains

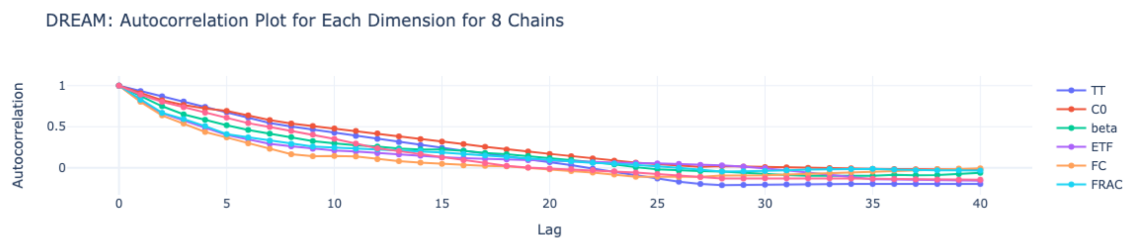


Figure 8.11.: Autocorrelation plot of the DREAM algorithm with 8 chains

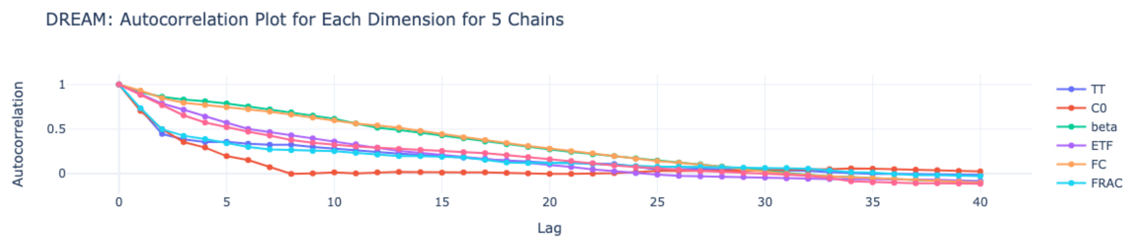


Figure 8.12.: Autocorrelation plot of the DREAM algorithm with 5 chains

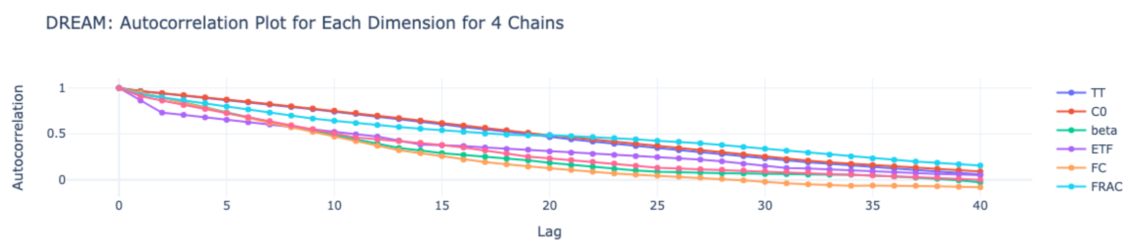


Figure 8.13.: Autocorrelation plot of the DREAM algorithm with 4 chains

8.2.5. Accuracy

The accuracy metrics including RMSE mean and MAE mean are also gathered for all test cases. The line plot of the RMSE mean displayed in 7.14 shows irregularity of the accuracy of the metric, with the accuracy scores for each test case being very close to each other. For the line plot of the MAE mean displayed in 7.15, a clear ascending pattern can be found. The more chains there are, the more accurate the Bayesian inference will be, though by a small difference.

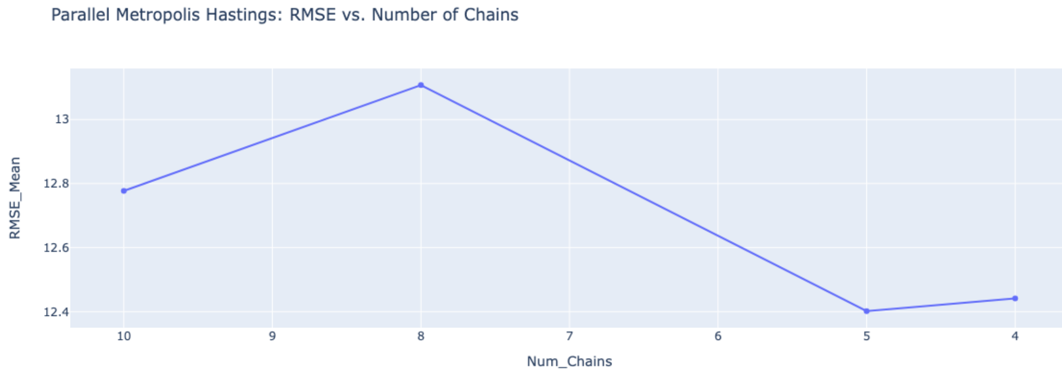


Figure 8.14.: Mean RMSE of the DREAM algorithm across test cases with different chains

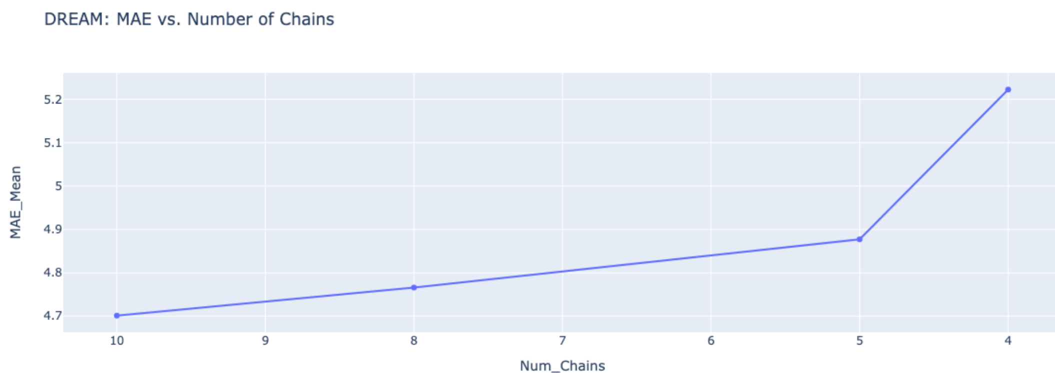


Figure 8.15.: Mean MAE of the DREAM algorithm across test cases with different chains

8.2.6. Parameter Overview

Moving on to the last section of the chain analysis, which is the parameter overview. Like the parallel Metropolis-Hastings algorithm, both distribution plots and boxplots are shown here. The focus here is put on the case of 10 chains and 4 chains, both extreme cases.

From the distribution visualization displayed in Figures 8.16 and 8.18, most of the chains for the same parameter have a peak in the region where the most samples are generated in the combined sample collection, with a few exceptions. For instance, for the TT parameter of the case of 10 chains, all of the chains display a peak near the higher bound for its sample space, which corresponds to the peak at the same position for the combined sample space. This is also the case for the C0 parameter for the test case of 10 chains, with the peak situated at around the position of 25% quantile. The exception here is the 7th chain, which is the only chain among all that does not have a peak there. The above-described scenario is opposite to the case of the parallel Metropolis-Hastings algorithm, in which each chain explores different areas of the parameter space. We can therefore find out the strong correlation between the sampling for each single chain and combined sample space.

For the boxplot displayed in Figures 8.17 and 8.19, however, the visualization shows no pattern at all. For some parameters like ETF from the test case of 4 chains or K2 from the test case of 10 chains, the medians across all chains are at around the same position. However, for the majority of cases, absolutely no regularity can be found to show where the positions of the 25% quantile, the median, and the 50% quantile are. Therefore, further investigation of the boxplot is not necessary.

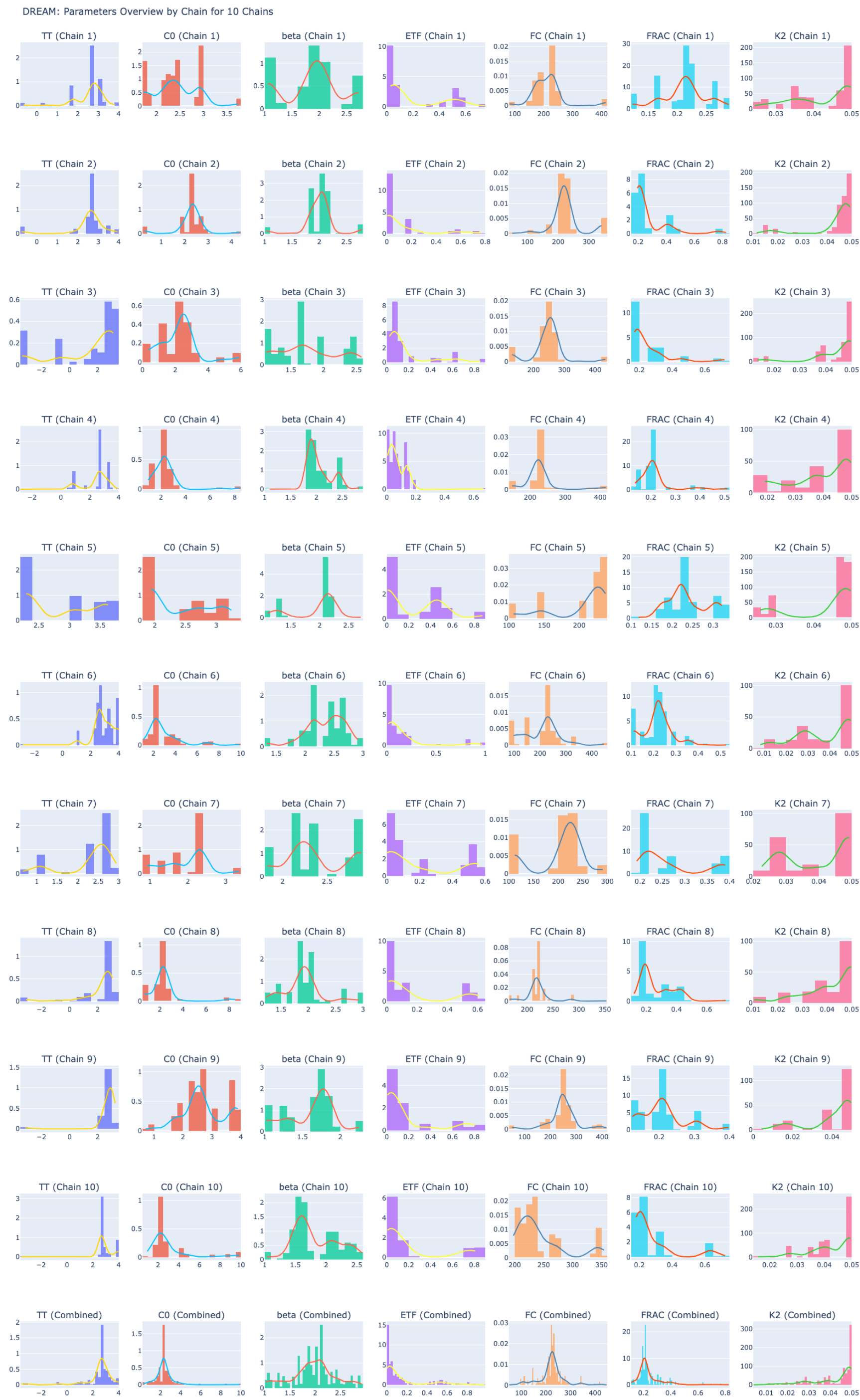


Figure 8.16.: Parameter overview by chain for DREAM using 10 chains

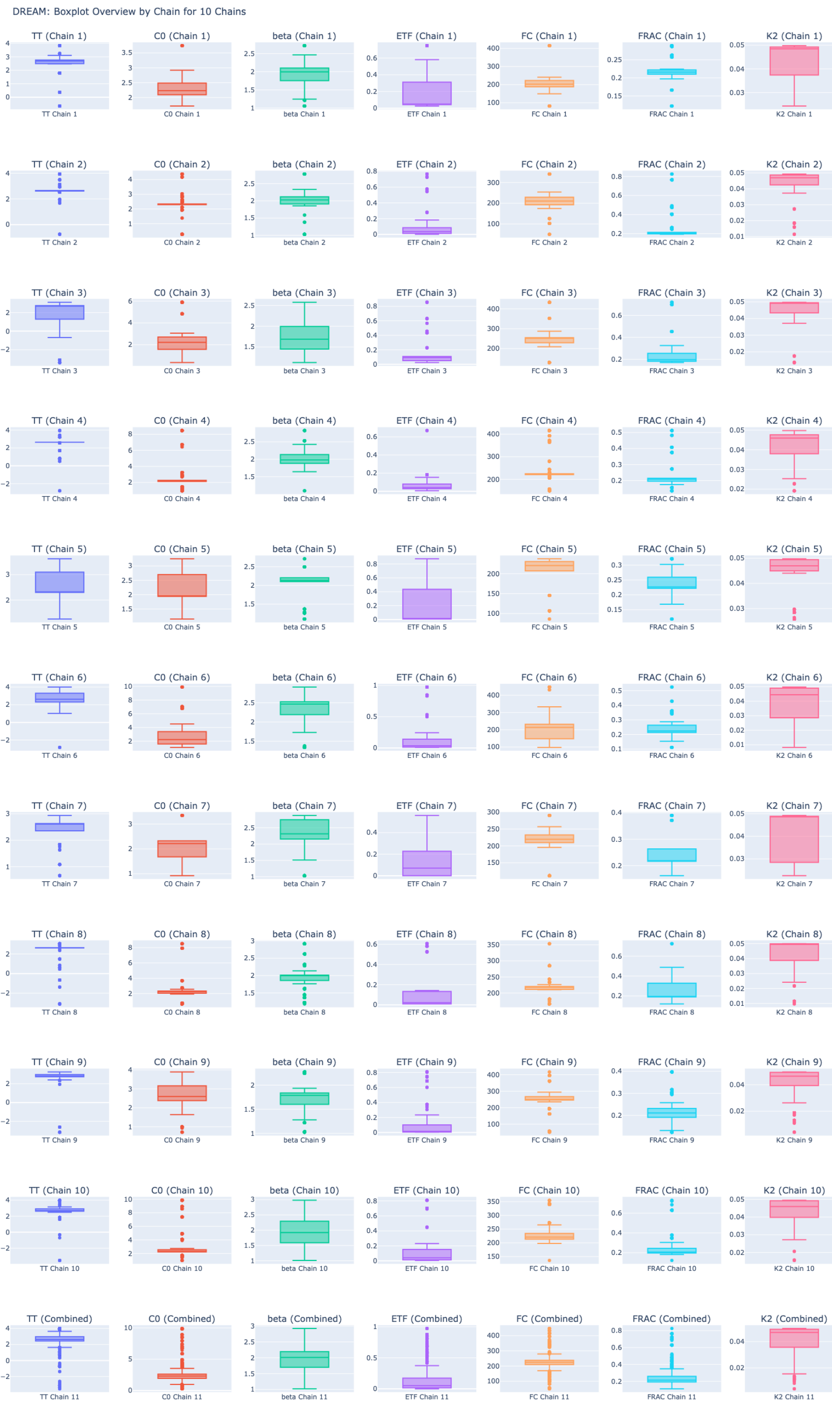


Figure 8.17.: Boxplot by chain for DREAM using 10 chains

Figure 8.18.: Parameter overview by chain for DREAM using 4 chains

Figure 8.19.: Boxplot by chain for DREAM using 4 chains

8.3. Input Algorithm Parameters Exploration

The last part of this chapter is the input algorithm parameter exploration for the DREAM algorithm. Being different from the other Markov chain Monte Carlo algorithms, the DREAM algorithm has a few input algorithm parameters that are unique to itself, with another few that are identical to the ones that the other Markov chain Monte Carlo algorithms possess. Explanation alongside analysis of benchmark data are listed below in smaller sections.

8.3.1. Sampling out of Bounds

The first input algorithm parameter that is investigated is the handling of the sampling out of bounds, which also exists for all of the other algorithms mentioned in this thesis.

For the DREAM algorithm, this parameter is called HardBoundaries, which determines whether the samples that are generated out of bounds are going to be reflected or ignored. The result is listed in Figure 8.20, where both results show relatively close accuracy and efficiency scores to each other. Even though not by much, the method of reflection performs generally better than the method of ignoring, which makes it a better choice for most use cases of the DREAM algorithm.

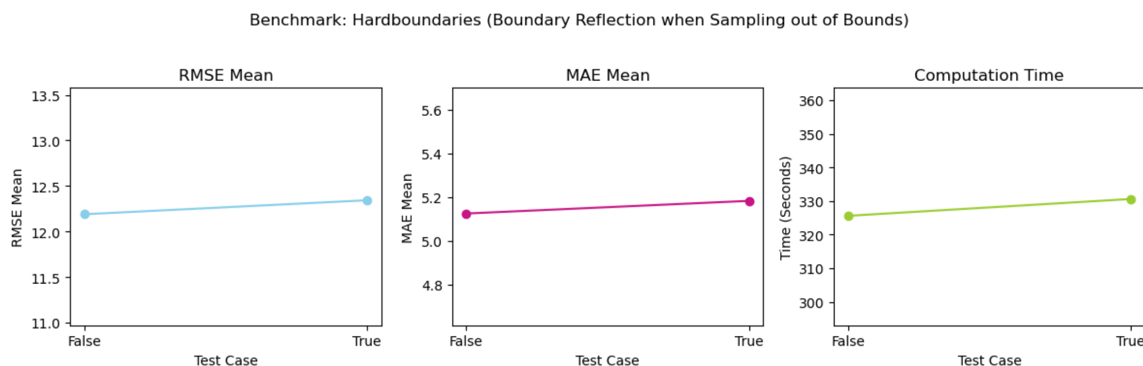


Figure 8.20.: Comparison of the accuracy and the efficiency of DREAM algorithms based on the HardBoundaries parameter

8.3.2. Crossover

In the DREAM algorithm, the concept of crossover is adapted from evolutionary algorithms and specifically implemented to enhance the proposal mechanism in comparison to other Markov chain Metropolis-Hastings algorithms. It is a process of combining information from multiple chains to create new proposal candidates, during which the components of the proposal vector are selectively swapped with corresponding components from other chains based on a crossover probability [**?**]. This method helps the algorithm explore the parameter space more efficiently by utilizing differences between chains.

The crossover burn-in is one of the aspects of the crossover concept. It denotes the number of iterations to fit the crossover values, ensuring the algorithm sufficiently adjusts and optimizes the crossover probabilities for effective parameter space exploration. The default value of this input algorithm parameter in PyDREAM is 10%. For testing purposes, however, the algorithm is also run with 0% (which is denoted as NaN in PyDREAM), 20%, and 50%. From the figure shown in 7.21, however, not much differences between the metric scores of all of the configurations are shown, apart from the slightly worse efficiency score of the 0% case against all of the other cases. This input algorithm parameter has, therefore, not that much influence on the Bayesian inference result.

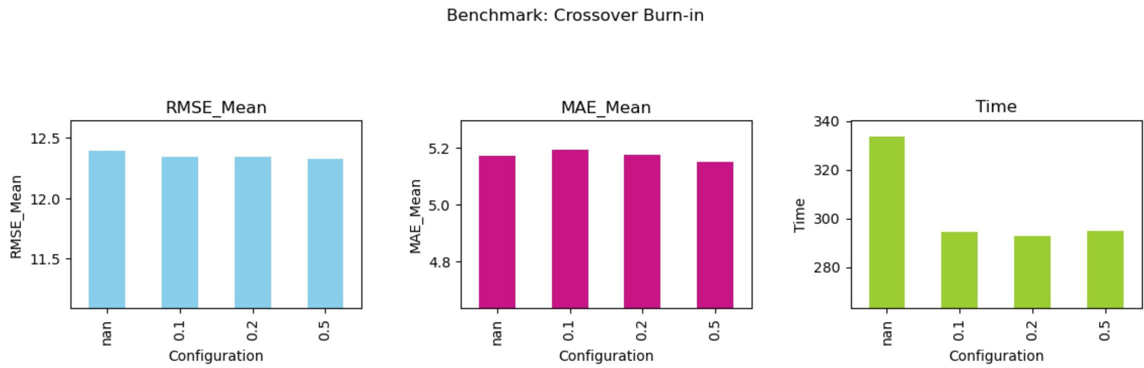


Figure 8.21.: Comparison of the accuracy and the efficiency of DREAM algorithms based on the crossover burn in parameter

Another aspect would be the adaptive crossover, which is responsible for the decision to adjust the crossover probabilities based on the performance of the chains [?]. This adaptation helps maintain an optimal balance between exploration and exploitation of the parameter space. By default, this option is set, even though we can also turn it off. The algorithm is therefore run in both variants, with the benchmark visualization displayed in Figure 8.22. However, there is only minimal difference between the metric scores of both variants, which is completely neglectable.

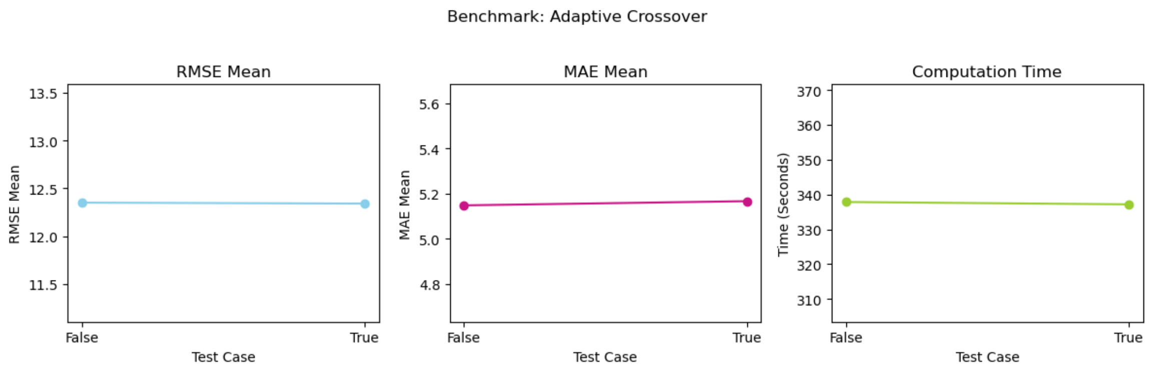


Figure 8.22.: Comparison of the accuracy and the efficiency of DREAM algorithms based on the adaptive crossover parameter

For the last aspect of the cross-over, we observe the nCR input algorithm parameter, which defines the number of crossover values to sample from during the run and to fit during the crossover burn-in period. Its default value is set as 3 for PyDREAM, whereas we also test two other cases, namely 1 and 3. From Figure 8.23, the differences between all these three cases are also neglectable, just as adaptive crossover. However, the default value of 3 generally provides worse metric scores, both in terms of accuracy and efficiency.

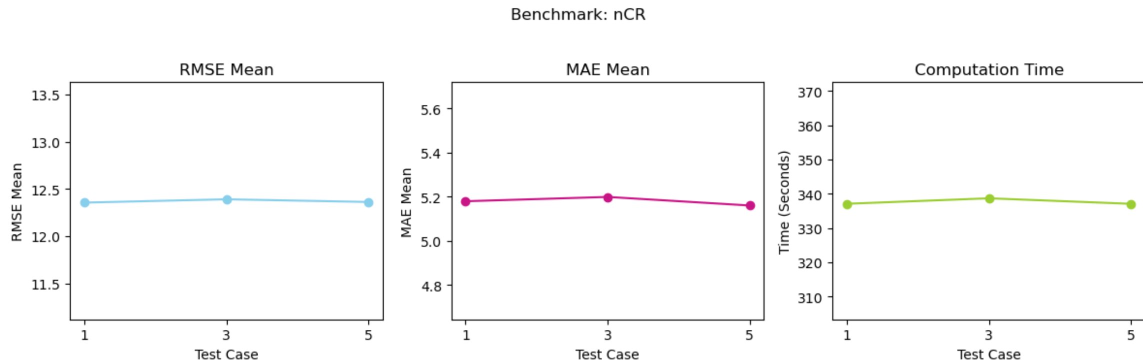


Figure 8.23.: Comparison of the accuracy and the efficiency of DREAM algorithms based on the nCR parameter

8.3.3. Likelihood Kernel

The likelihood kernel function is manually defined for the DREAM algorithm, just as any other Markov chain Monte Carlo algorithm. An investigation here is therefore also necessary. We keep the likelihood kernel in the same format as the one used in other algorithms, with two available options that include independent and dependent versions.

For the dependent version, the best accuracy performance happens at low likelihood kernel factors, with the metric inaccuracy growing as the factor grows, even though the differences are also neglectable. However, the computation time of the factor 5 is the most optimal, being almost 20 seconds faster than other test cases. This efficiency difference could play an important role in the selection of value.

For the independent version, the best accuracy performance happens at 0.6. There are, however, no patterns that can be found. For the computation time, the efficiency grows as the factor value grows. However, the independent version of the likelihood function performs much worse than the dependent version of the likelihood function, both in RMSE and in MAE metrics. Therefore, this version of the likelihood function is not considered in the case of the DREAM algorithm.

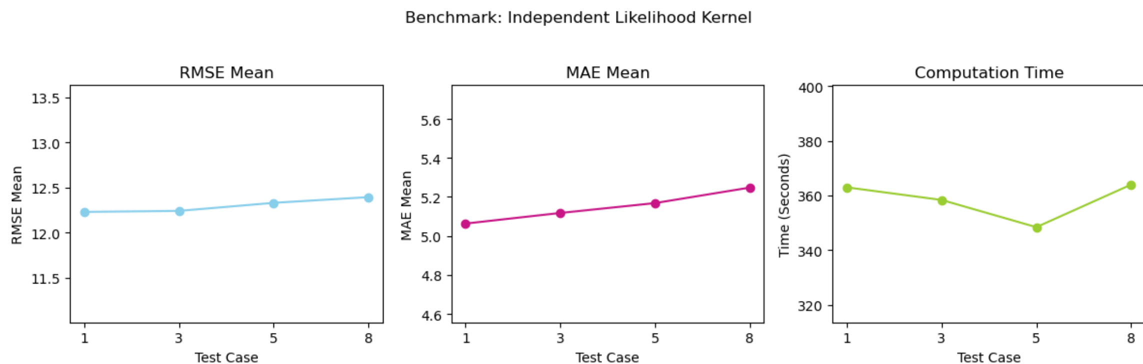


Figure 8.24.: Comparison of the accuracy and the efficiency of DREAM algorithms based on the independent likelihood kernel factor

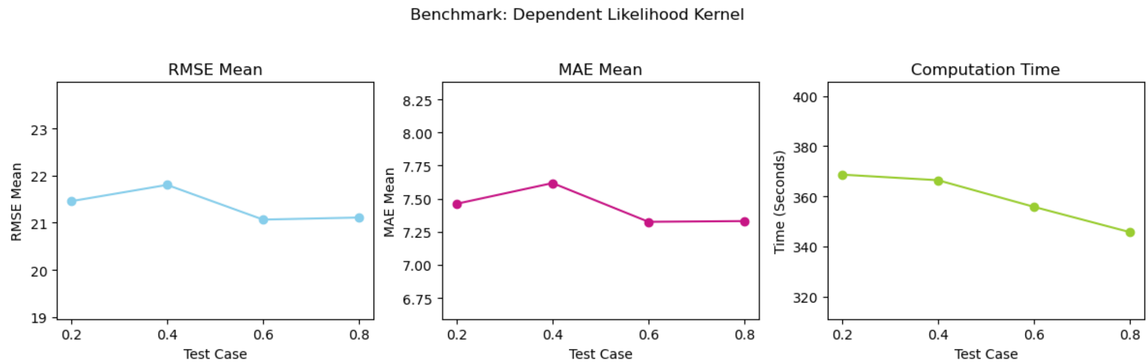


Figure 8.25.: Comparison of the accuracy and the efficiency of DREAM algorithms based on the dependent likelihood kernel factor

8.3.4. Initialization

The initialization method is a big topic for efficiency enhancement. It is no exception for the case of the DREAM algorithm. The same initialization methods as other algorithms are proposed and used here. The visualization is then displayed in Figure 8.26. As expected, the accuracy metrics do not differ much from each other. However, the most efficient initialization methods according to the efficiency metrics are lower bound, upper bound, 1st quantile of the prior distribution, the mean of the prior distribution, 3rd quantile of the prior distribution, and the 1st quantile of the posterior distribution.

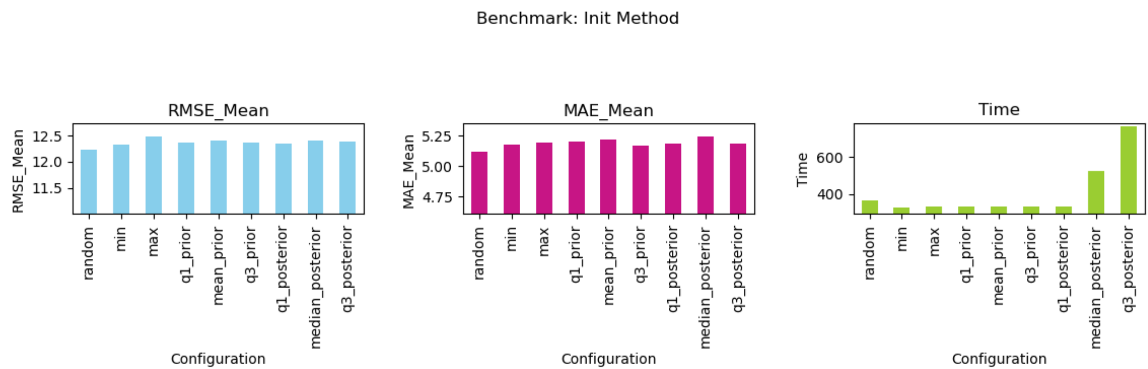


Figure 8.26.: Comparison of the accuracy and the efficiency of DREAM algorithms based on the initialization method

8.3.5. Gamma

In the DREAM algorithm, the parameter gamma controls the step size in the proposal generation process, which determines how far the new proposals can move from the last sample. When gamma equals 1, the updates are larger, which enables the algorithm to make broader moves, allowing the algorithm to avoid local extreme points and improve the chain mixing. In PyDREAM, the parameter to adjust is called p_gamma_unity, which specifies the probability that gamma will be set to 1 during the sample generation. By adjusting it,

we can balance between exploration with larger steps and exploration with smaller steps, making the sampling process able to be customized for specific use cases and requirements. The default probability of p_gamma_level is 20%, whereas all values between 0 and 1 with a distance of 0.2 are used for testing purposes. Figure 8.27 documents the benchmarked data in a visual way, from which we can directly infer that the default value of 0.2 and the other value of 0.8 deliver good RMSE scores, where 0.8 outperforms the 0.2 case by MAE, though not by much. For the run time, however, there is indeed a noticeable difference, where the configuration of 0, 0.2, and 1 all deliver a more ideal efficiency score than other test cases.

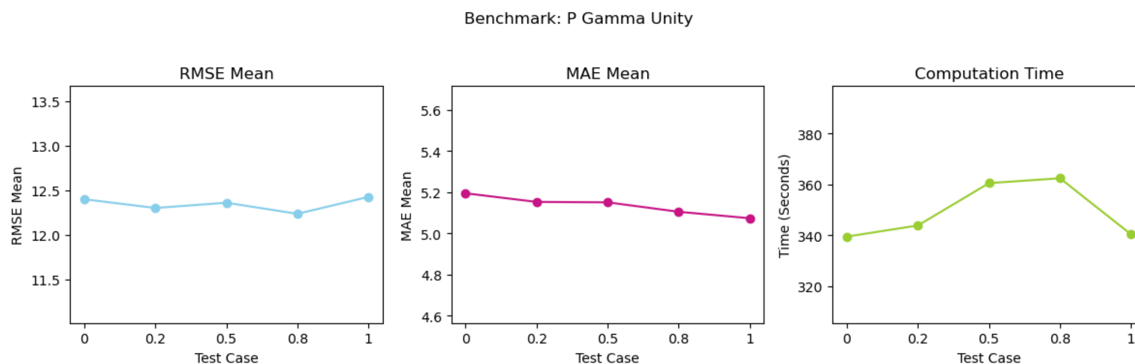


Figure 8.27.: Comparison of the accuracy and the efficiency of DREAM algorithms based on the p gamma unity parameter

8.3.6. Differential Evolution

Differential Evolution (DE) is an optimization algorithm that iteratively enhances a population of candidate solutions by generating new candidates through the weighted difference between pairs of existing solutions (DEpairs) and combining them with a third solution. In the DREAM algorithm, DEpairs ensure diverse and effective exploration of the parameter space, while the snooker update further enhances exploration by projecting vector differences onto the current state, helping to navigate complex, multimodal distributions. Together, these mechanisms enable robust and efficient optimization in high-dimensional problems.

In the DREAM algorithm, differential evolution is an optimization method that improves the sampling candidate solutions by generating new samples using the weighted difference between pairs of existing samples and then using the crossover to mix components of these solutions. Using this method, provides robustness for the algorithm, ensuring it can explore complex and high-dimensional parameter spaces efficiently. To tune the DREAM algorithm with respect to differential evolution, there are two input algorithm parameters available. These are discussed in the following subsections.

DEpairs

In the introduction part, the concept of using the weighted difference between pairs of existing samples is mentioned. The amount of sample pairs is called DEpairs, which is responsible for generating new samples that keep maintain the diversity in the sample result and the good mixing of the chains. By default, only one pair of existing samples is chosen

for the calculation. For testing purposes, other values including 2 and 3 are tested, so that we can observe to which extent the number of pairs affects the Bayesian inference result. The benchmark data is recorded as visualization in Figure 8.28, where we can infer that using 2 pairs of existing samples provides slightly better accuracy both in terms of RMSE and MAE than the rest of the pair amounts. The computation time is, however, a complete reflection of the accuracy metrics, where using 1 or 3 pairs of existing samples provides an overall better efficiency than the case of 2. Thus, the selection of value here is a trade-off between accuracy and efficiency.

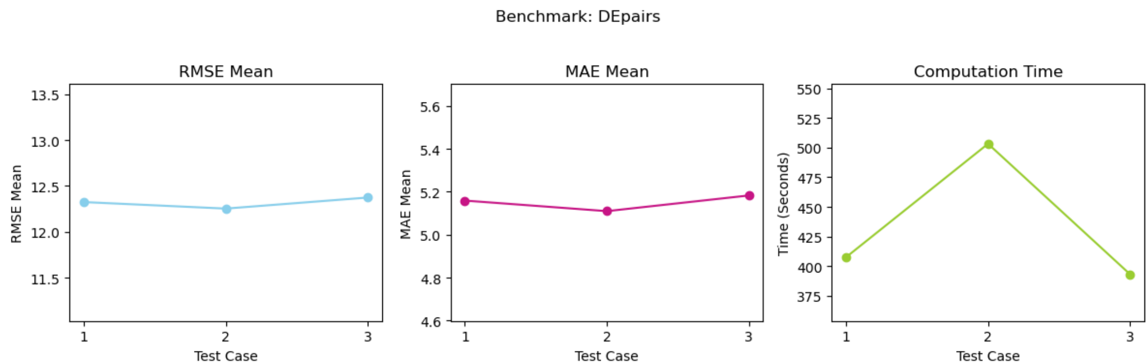


Figure 8.28.: Comparison of the accuracy and the efficiency of DREAM algorithms based on the DEpairs parameter

Snooker

Another mechanism in terms of differential evolution that is applied in the DREAM algorithm, other than the above-mentioned pairs inference, is the snooker update. It further enhances exploration by projecting vector differences between two chains onto the current state instead of randomly selecting pairs for further sampling. In the DREAM algorithm, a probability is set for the algorithm to determine whether to use a snooker update instead of a regular update for the next iteration, since a snooker update is more compute-heavy. The default probability is set as 10%, whereas we test different values including 20%, 50%, and 80%. Besides, the two extreme cases are also tested, where we completely ditch the idea of snooker update and only keep the regular updates based on randomly selecting pairs (0%), and where we only use snooker update in each iteration (100%). The benchmark data is stored in Figure 8.29. For efficiency, there is not much difference between the run time across all configurations. For the accuracy metric, on the other hand, the extreme case of not using the snooker update delivers the best performance in comparison with other cases, where the snooker update is partially or completely used. The conclusion is then drawn, that for the use case of the hydrological model, not using the snooker update delivers a more accurate result.

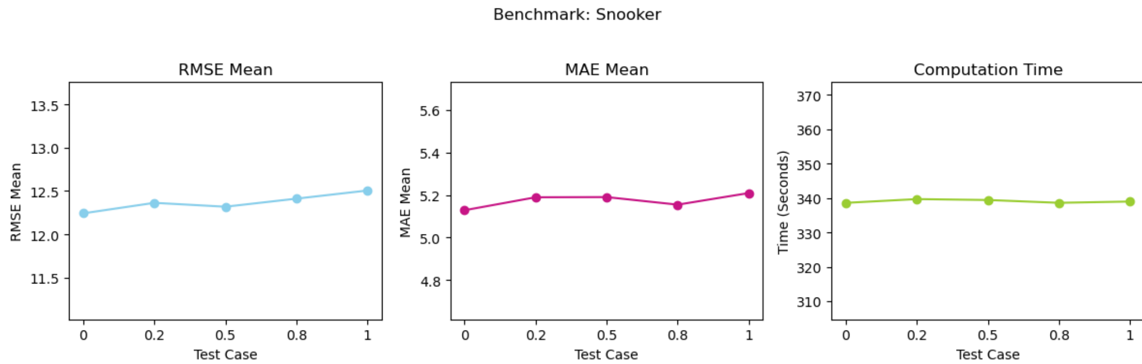


Figure 8.29.: Comparison of the accuracy and the efficiency of DREAM algorithms based on the snooker parameter

8.3.7. Burn In Phase

Burn-in phase is another crucial topic for all Markov chain Monte Carlo algorithms. From the analysis in the chapters above, the burn-in phase poses a great influence on the outcome of the Bayesian inference, therefore it is also investigated here. The benchmark result is visualized in Figure 8.30. For the aspect of accuracy, we pick the factor over 5, which is the 20% burn-in phase. The relationship between the efficiency metrics and the configuration is, on the other hand, the complete opposite of the relationship between accuracy metrics and the configuration. The lower factor, which is the 50% burn-in phase, results in a more efficient run time than the higher factor, which is the 20% burn-in phase. The selection of the burn-in phase amount then also becomes a trade-off between accuracy and efficiency.

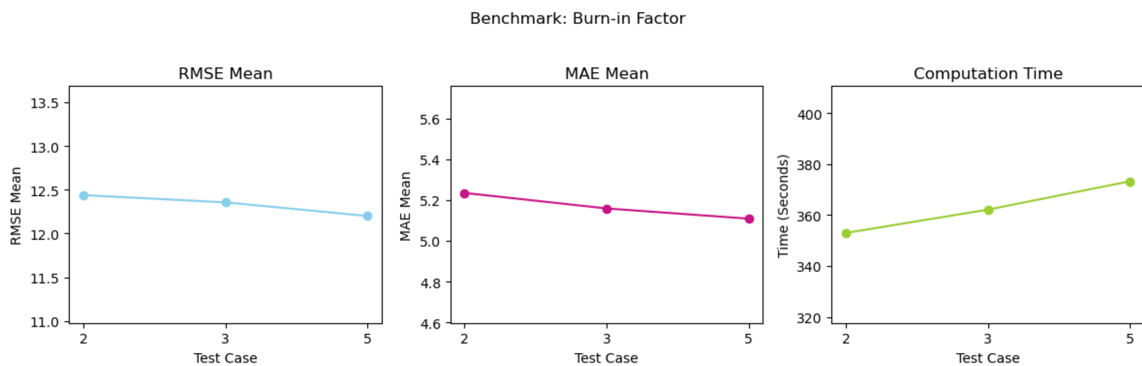


Figure 8.30.: Comparison of the accuracy and the efficiency of DREAM algorithms based on the burn in factor

8.3.8. Effective Sample Size

Unlike the burn-in input algorithm parameter, the effective sample size shows no patterns that can be found. From the graph shown in Figure 8.31, the only information that is presented is the most optimal configuration for accuracy score, namely the effective sample

size of 5, and also the best configuration in terms of efficiency, namely the effective sample size of 2, 4 and 5.

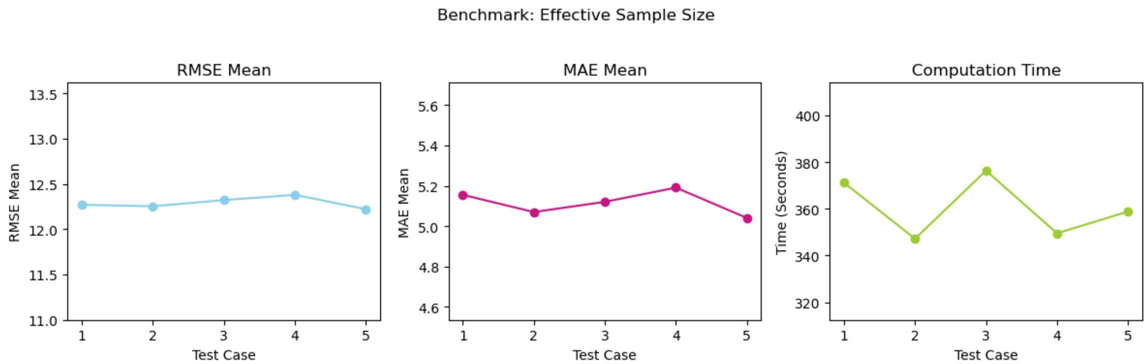


Figure 8.31.: Comparison of the accuracy and the efficiency of DREAM algorithms based on the effective sample size

8.4. Result and Comparison with Other Algorithms

Same as the case for the general parallel Metropolis-Hastings algorithm, the dream algorithm is also run one more time, so that we can gather the data output for visualization. A comparison with the other algorithms is going to be made, both in terms of accuracy and efficiency.

The overview of the posterior distribution after the calibration is shown in the first graph, where the generated samples are more concentrated than in the posterior generated by the general parallel Metropolis-Hastings algorithm. This suggests that the DREAM algorithm has a higher efficiency in exploring the parameter space, leading to more precise estimations of the posterior distributions. The increased concentration of samples suggests a more reliable convergence and leads to better efficiency of the calibration process. Altogether with the concentration of the aggregation of generated samples from each dimension, the DREAM algorithm is the best choice in terms of the parameter calibration under uncertainty, exceeding the general parallel Metropolis-Hastings algorithm.

Figure 8.32.: Overview of the posterior distribution of the parameters calibrated by the general parallel Metropolis-Hastings algorithm

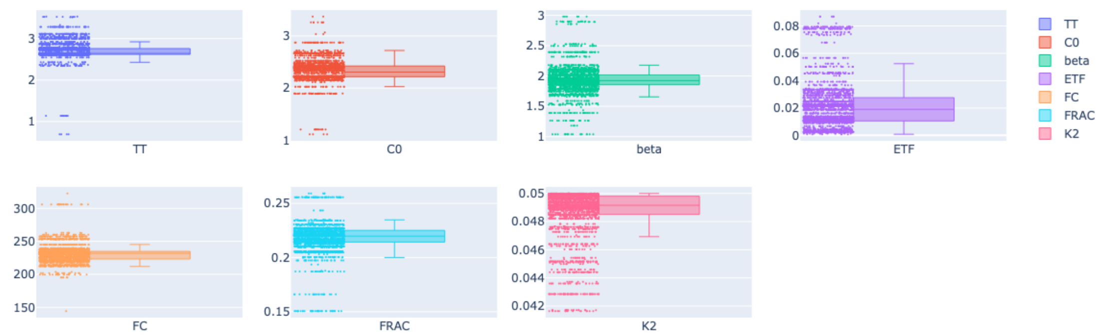


Figure 8.33.: Boxplots of the generated posterior samples of each parameter calibrated by the general parallel Metropolis-Hastings algorithm

In terms of accuracy, the RMSE score of the algorithm is around 20 and the MAE score is around 9 for most cases, which is a slight downgrade from the general parallel Metropolis-Hastings algorithm. Considering the extreme boost regarding the efficiency, with an average runtime of around 400 seconds against 4000 seconds, the DREAM algorithm would be an optimal choice for the uncertainty quantification.

9. Sampling Time Series Analysis

In this chapter, a deeper look into the time series is given. Different time series for training and testing purposes are selected and used for the performance of Bayesian inference so that the inferred results can be compared to each other and the measured data.

9.1. Sampling and Testing Time Series Selection

To set up the analysis, a range of time series that are used for sampling and testing need to be selected. The sampling time series is used on which the Markov chain Monte Carlo algorithms could perform sampling. In other words, the posterior, which is the result of the Markov chain Monte Carlo sampling, is trained based on these data. The generated posterior is then used for Bayesian inference, in which the Monte Carlo simulation is used for random sampling and used as input for the HBV-SASK model. The result generated is then compared with the measured data by visualization so that the differences between the two time series can be observed.

The selection of these times series requires generality, which means that cases of different scenarios need to be taken into consideration. Therefore, a deep look into both datasets provided alongside the HBV-SASK model needs to be conducted. The visualization with times series decomposition is displayed in Figures 3.1 and 3.2. From these visualizations, we figure out that anomalies are integral parts of the entire time series, as they provide valuable information that can be analyzed. These anomalies could be led by different reasons, including climate changes, human activities, and seasonal variations. The predominant reason, however, is flooding [**?**], which occurs now and then according to the visualizations. They are therefore heavily taken into consideration in terms of sampling and testing times series selection.

For the Oldman Basin, the anomalies are relatively apart from each other. From the trend, we can see that relatively even intervals exist between different occurrences. The entire time series is relatively calm and balanced in comparison to the Banff Basin, indicating the seasons with high levels of discharge and raindrops are more predictable and less extreme. Nevertheless, periods between the years 1992 and 1996 do have higher peaks across the entire times series, with a significant peak presenting in June of 1995. Other small amounts of lower peaks are distributed across the whole time series. The calmness with a certain amount of anomalies makes the Oldman Basin data set an optimal choice for the testing of generality. Therefore, most of the time series that are used for training and testing are selected from the Oldman Basin dataset.

For the Banff Basin, a completely different scenario is presented. High peaks exist everywhere and very often, causing the contrast between the flood period and the drought period to be significant. The data is constantly fluctuating, showing active climate changes and constant floods that are recorded in the area. The trend of the time series also displays no patterns. These characteristics make it an optimal dataset for extreme case prediction, is

therefore used for edge cases testing, and has less time series included for testing or training purposes.

To cover the generality of the time series behavior, the following data sets are selected. For training, the following time series are selected:

- Short and Calm (Oldman): A short and calm time series. We intend to observe whether the lack of anomalies has an impact on the final inferred result.
- Short with Peaks (Oldman): A short time series with peaks that represent potential anomalies. We intend to observe how the small amount of data with extreme anomalies affects the outcome of the Bayesian inference.
- Long and Calm (Oldman): A short and calm time series. Alongside the lack of anomalies, the necessity of having a huge amount of data is at the center of the observation.
- Long with Peaks (Oldman): A long times series filled with anomalies. This data frame represents the most generality, as it includes both calm periods and anomaly periods.
- 97-03 (Oldman): A time frame that lies inside of the Oldman Basin dataset to represent generality. The years correspond to these of the time series ”97-03 (Banff)” for ease of comparison.
- 97-03 (Banff): A time frame that lies inside of the Banff Basin dataset to represent generality. The years correspond to these of the time series ”97-03 (Oldman)” for ease of comparison.

For testing, the following time series are selected:

- Data containing floods (Oldman): The posterior is tested on a time series where anomalies that represent floods are present. The ability to predict anomalies is tested.
- Data displaying calmness (Oldman): The posterior is tested on a calm time series. The ability to cope with calm time regions with less fluctuations is tested.
- Banff: A general Banff sub time series with fluctuations that represent the generality of the Banff time series is also used for testing, specifically for the posterior that is trained using the Banff data set.

Another important factor for the execution of the HBV-SASK model is the spin-up phase. The spin-up phase is performed before the actual execution of the model, where the model performs execution based on the time series that are in other time frame than the time frame of the actual data, usually a certain period before the period of the actual data. It is performed to ensure the reduction of Initial Condition Bias and temporal consistency [**?**]. A spin-up period of 400 cycles is suggested [**?**], however, we intend to test the impact of the spin-up phase on efficiency and accuracy metrics, since the lengths of each time series that are used for training are inconsistent. Therefore, 25%, 50%, and 100% of the time, which is the time frame of the actual data, will be tested for the spin-up phase.

9.2. Result Interpretation

After extensive analysis, we conclude that the results of all different training data sets do not show significant differences from each other. The same patterns are shown across all

posterior samples from different training data sets. However, there are a few details that show the characteristics of the training data set, which can be observed in the visualizations. In the following paragraphs, the two testing scenarios are interpreted separately. The first four training time series, namely short and calm, short with peaks, long and calm alongside long with peaks are discussed to offer more insights into details, while the rest of the time series are going to be discussed in the next chapter.

For the test case on data containing floods (Oldman), all inferred results shows decent performance, particularly regarding trends. The posterior mean aligns well with the measured data, since it closely follows the measured data throughout the entire period, both calm periods and periods with high peaks. This indicates that the algorithm is capable of accurately estimating flood activities and anomalies. On the other hand, the inferred result shows more fluctuations than the measured data, suggesting that the algorithm is sensitive to small changes in the input data or noises.

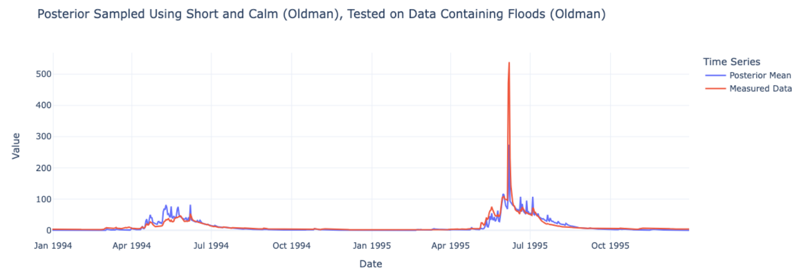


Figure 9.1.: The Bayesian inferred result of the posterior sampled using the short and calm time series, testing on data containing floods

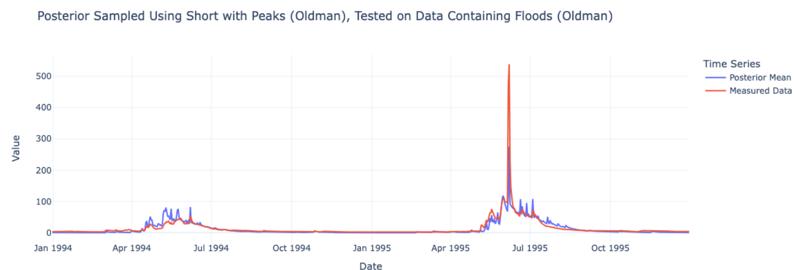


Figure 9.2.: The Bayesian inferred result of the posterior sampled using the short with peaks time series, testing on data containing floods

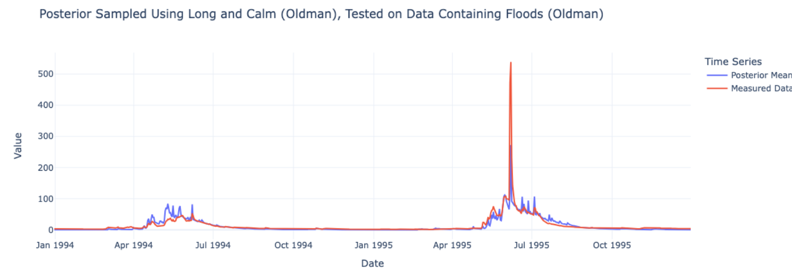


Figure 9.3.: The Bayesian inferred result of the posterior sampled using the long and calm time series, testing on data containing floods

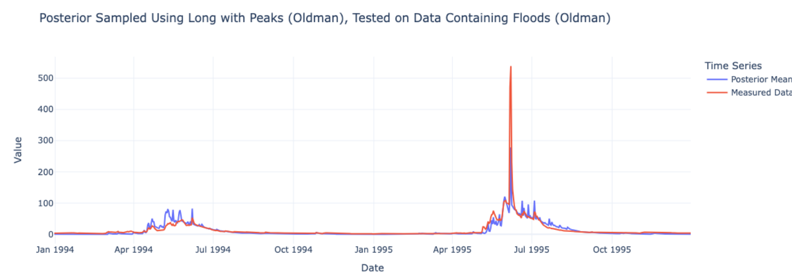


Figure 9.4.: The Bayesian inferred result of the posterior sampled using the long with peaks time series, testing on data containing floods

A challenge here is the prediction of the peak in June 1995. The peak indicates a potential strong flood, which is a significant anomaly across the entire time series. For comparison, the exact predicted value is gathered manually from all of these scenarios. These are presented in the table down below.

Training Data	Short Data Frame	Long Data Frame
Calmness	272.6607	272.9323
With Peaks	275.1849	277.3239

The measured result is 539, which is far away from the inferred data. Nevertheless, some level of dependencies can be found here, as the training time series with peaks perform better than the ones without peaks. This observation is logical since the posterior sampled from the time series with peaks is more used to anomalies and peaks across the entire time series. On the other hand, the inferred result from the long data frame performs slightly better than the inferred result from the short data frame. This is potentially due to the same reason, where the posterior sampled from the long period might be exposed to more anomalies than the one sampled from the short period.

For the test case on data displaying calmness (Oldman), more discrepancies are shown in the visualization due to the high scale-in level. The inferred results generally follow the trend

of the measured data, especially in the calm periods. In fluctuating periods, the inferred results also fluctuate, with peaks not being predicted as the exact measured value. This high-variable property proves that the algorithm is susceptible to fluctuations, as inferred in the test scenario above. A visualization of the first test case is displayed below. Other scenarios offer similar results to the first test case in terms of visualization and are therefore not presented here. Further comparisons regarding accuracy and efficiency metrics are discussed in the next section, namely result comparison.

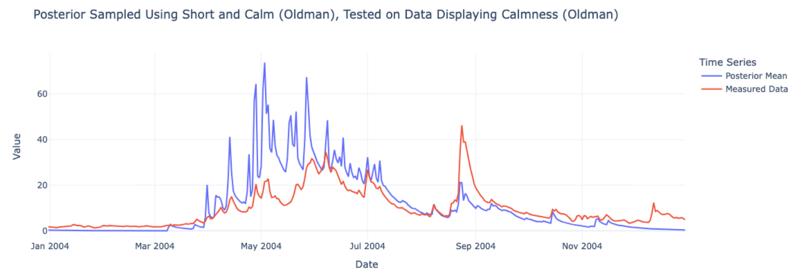


Figure 9.5.: The Bayesian inferred result of the posterior sampled using the short and calm time series, testing on data displaying calmness. All of the test cases on data displaying calmness share similar results to this visualization

9.3. Result Comparison

After visualizing the results and comparing them with the measured data, we quantify the accuracy and efficiency using metrics. In the last section, the test mentioned above scenarios will be compared with each other in pairs, so that the dependency between metrics and specific properties of training data sets can be visualized.

The first comparison takes place between the training time series of short periods. The short and calm time series is compared with the short with peaks data frame, so that the factor of including anomalies for the sampling phase of the Markov chain Monte Carlo algorithm is investigated. From the bar chart, the short and calm time series seems to deliver all-around better performance than the training dataset containing peaks. This could be because the majority part of the time series is calm, and sampling the posterior from the period that mostly comprises calmness allows the posterior to get accustomed to the generality of the data during the sampling phase. On the other hand, the conclusion that is drawn in the section above regarding the peak prediction is still valid. The posterior sampled from the time series that contain peaks are more accustomed to the anomalies and are therefore able to make better predictions for peak areas.

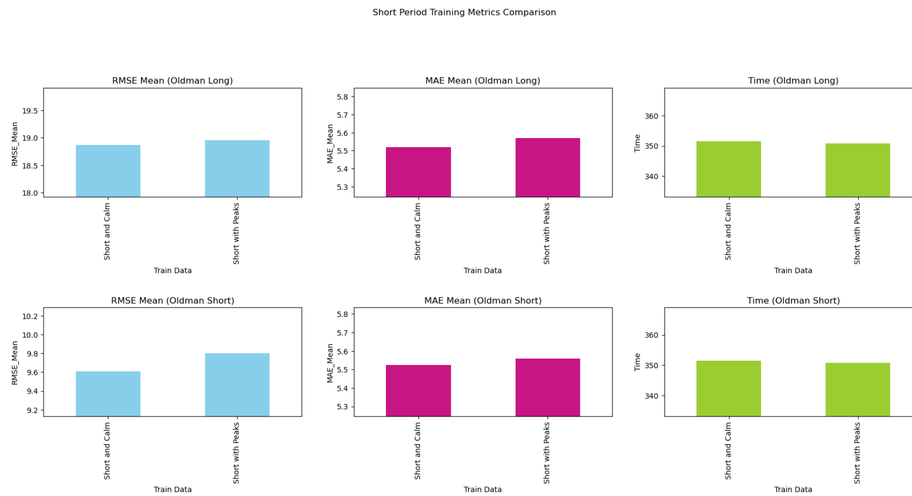


Figure 9.6.: Comparison of metrics for the test case of short periods

The second comparison is between training time series of long periods. Logical relationships are also derived in this case, where the posterior sampled from the long time series with peaks, where anomalies occur now and then, delivers better performance. Having appropriate amounts of anomalies in the training data set, the posterior can resemble the regularity of presence in terms of patterns of anomalies, while it keeps the model robust and accurate in predicting calm phases. This balance helps the model to better generalize and adapt to unexpected variations in the data.

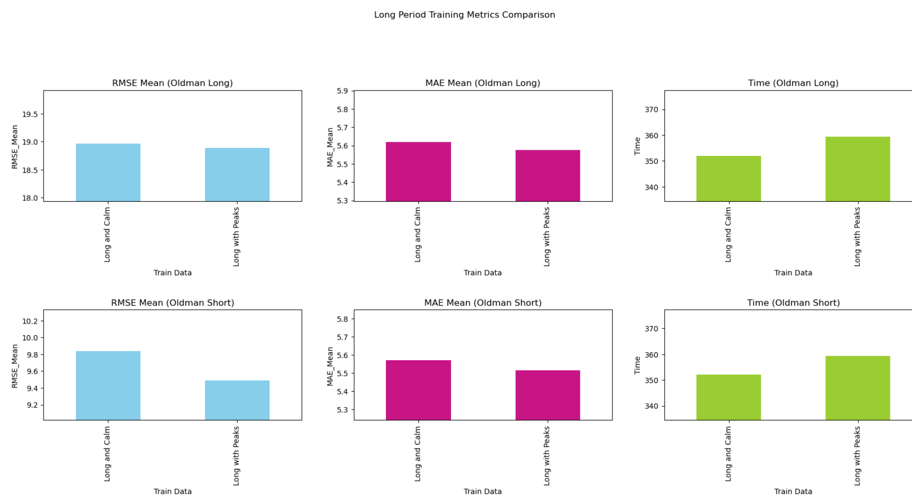


Figure 9.7.: Comparison of metrics for the test case of long periods

For the third comparison, we focus on periods with peaks. A short time series with peaks is compared with a long time series with peaks so that the importance of the training period length is focused. From the results, we can observe that the long training time series generally delivers a better performance than the short training time series. This observation is reasonable, since the long training time series contains more anomalies over time, allowing

the posterior to learn and adapt to the pattern in the process of sampling. A longer period for training time series here means that the posterior is exposed to more anomalies, allowing the adaption to take place, thus potentially resulting in better accuracy scores.

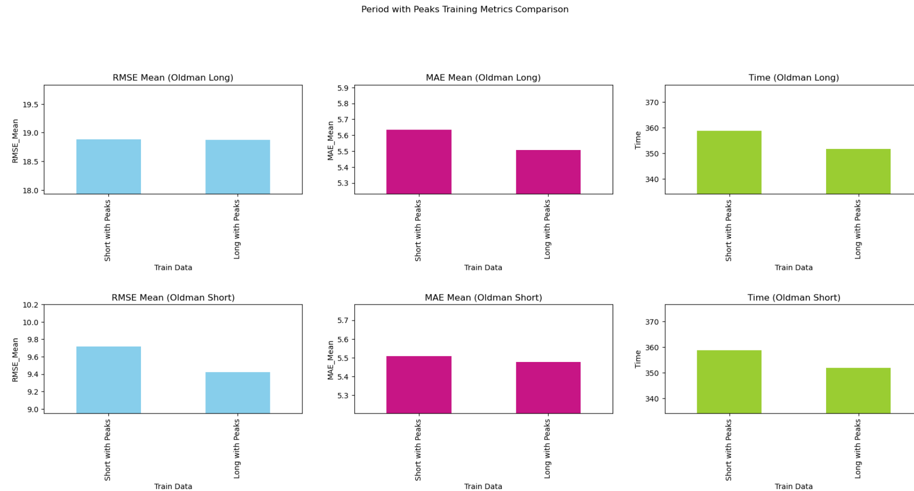


Figure 9.8.: Comparison of metrics for the test case of periods with peaks

The fourth comparison is more generalized. Time series between 1997 and 2003 of both Banff and Oldman basins, which contain both peak and calm phases, are used to generate posteriors, where they are then tested on two testing data frames selected from both basins. For the test data frame from the Oldman basin, the posterior sampled from the Oldman basin achieved a better accuracy score than the posterior sampled from the Banff basin. This is logical because the sampling process adapts itself to the behavior of the Oldman basin. However, the posterior sample from the Oldman basin also performs better for the test data frame from the Banff basin than the posterior sampled from the Banff basin. This could be due to the same reason as mentioned in the first comparison since anomalies are still. Being trained on the Oldman basin instead of the Banff basin, the posterior is more accustomed to the calm periods, which still constitutes the majority part of the time series from the Banff basin. Therefore, training on the Oldman basin is overall a better choice for general Bayesian inference results.

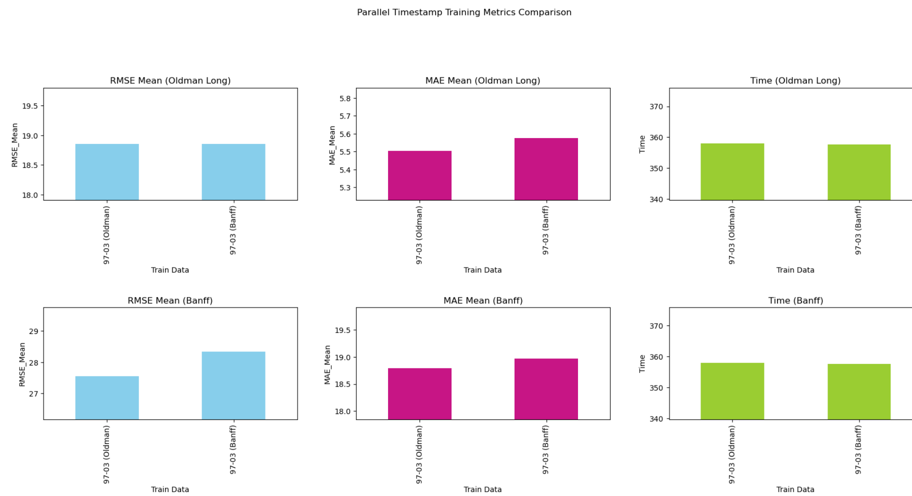


Figure 9.9.: Comparison of metrics for the test case of periods from different basins

The spin-up phase is investigated for the last comparison. The efficiency is more optimized for the models with shorter spin-up phases, which is logical since spin-up phases require model executions and simulations as well. For the accuracy metrics, on the other hand, the results show a certain complexity for analysis. Testing the sampled posterior on the short test data frame results in a proportional relationship, in which shorter spin-up phases result in better accuracy scores. This might be because longer spin-up phases consider too much historical data, which might potentially be a disturbing factor for the Bayesian inference. For the long test data frame, there is no pattern that can be distinguished regarding the relationship between the accuracy score and the spin-up length. Since the longer data frame contains more anomalies, the behavior of the posterior sampling can be unpredictable. Therefore, the spin-up length may not consistently influence the accuracy score.

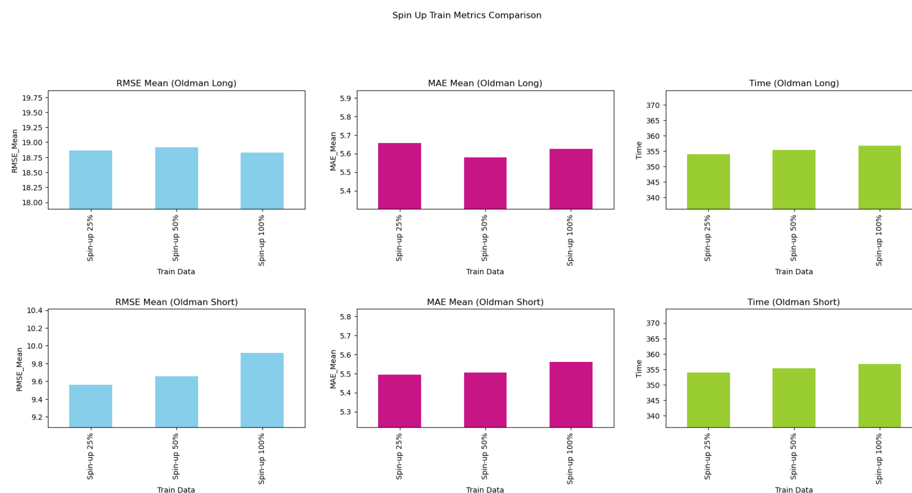


Figure 9.10.: Comparison of metrics for the observation of spin-up length

In conclusion, sampling from data where more peaks are present allows the Markov chain Monte Carlo algorithm to perform more precise predictions for anomalies, whereas sampling from data that are calmer and contain fewer anomalies allows the Markov chain Monte Carlo algorithm to perform generally better forecasts in terms of accuracy metrics. A selection of training data that balances both aspects is crucial for optimal inferred results.

10. Description of Software Implementation

In this chapter, a brief documentation of the software implementation of this thesis is given. An overview of the structure will be documented, as well as how to use the implemented framework to perform Bayesian inference using different algorithms.

10.1. Structure and Usage

The repository[1] contains three main subfolders. ”Thesis” includes the source file of the actual thesis and ”Results” includes the benchmarked data and visualization for the thesis. The most important folder regarding the actual software is ”Implementation”, where the implemented algorithm and the data are stored.

The structure of the algorithm is as follows: the main file that is executed is called ”run.py” under the src subfolder. This file is responsible for calling the selected implemented algorithm and executing the Bayesian inference with specified parameters. Specifying the arguments that configure the run time environment can be done by editing the file ”run_config.json” on the root level. This file uses the JSON format to configure the run-time parameters of the software. The requirements or options are documented below:

- configPath (required): The config file that is used for the model.
- basis (required): The basis of the data that is used for the model.
- mode (required): The mode of the algorithm. Options: mh, parallel_mh, gpmh, dream.
- separate_chains (optional, default=false): Determine whether the output file is supposed to record the data separately by chains. They are only relevant for algorithms that sample data using multiple chains, including parallel_mh and dream.
- burnin_fac (optional, default=5): The burn in factor that is used for the result of the MCMC algorithm. The first 1/burnin_fac percentage of the entire data is going to be discarded.
- effective_sample_size (optional, default=1): Only the every n_th data is going to be collected. Default 1: no data point is going to be discarded.
- output_file_name (optional, default=”mcmc_data.out”): The file name of the saved output result.
- kwargs (optional): A dictionary in form of JSON that is used for specific algorithm input parameters.

The first parameter, configPath, leads to another configuration file that is used for the hydrological model instantiation. This configuration file for the hydrological model is typically stored in the configuration subfolder under implementation.

[1] `https://github.com/CJZbeastmode/HBV-SASK-Bayesian-Inference`

After configuring the configuration file, we execute the software by running "run.py". The software then fetches the data from the configuration file, loading the model via model initialization functions, which are implemented in "construction_model.py" and "execute_model.py", and selecting the corresponding algorithm and likelihood function. The algorithm initialization files are stored in the folder run_mcmc, acting as a preparation for executing the actual algorithms, all of which are stored in the dependencies subfolder on the level before. The different implementations of likelihood functions are stored in the likelihood subfolder. The samples that are generated will be stored as a CSV file as output.

For the visualization part, the Jupyter notebook file "visualization.ipynb" is provided under the "src" folder. To configure the visualization, the file "viz_config.json" is used. Individual parameters for configurations are listed below.

- configPath (required): The config file that is used for the model.
- basis (required): The basis of the data that is used for the model.
- input_file (required): The data in the input file. It could be seperately recorded or merged.
- sep_viz (optional, default=False): The option to visualize the data by chains. If false, then the entire dataframe is going to be visualized. If true, different chains are going to be visualized individually, before a comparison visualization is going to be given.
- monte_carlo_repetition (optional, default=1000): The number of iterations for the monte carlo method for the comparison of the Bayesian inference result.

10.2. Algorithm Specification

As mentioned in the parameter explanation in the last section, "kwargs" indicate specific configurations for the Markov chain Monte Carlo algorithms. In this section, details regarding these specifications are documented.

Configurations for the fundamental Metropolis-Hastings algorithm include:

- version (optional, default="ignoring"): Version of the MH algorithm. Options: ignoring, refl_bound, aggr.
- sd_transition_factor (optional, default=6): The standard deviation factor of the transition kernel. The standard deviation is given by (upper bound - lower bound) / sd_transition_factor.
- likelihood_sd (optional, default=1): The standard deviation parameter for independent likelihood function, or the standard deviation parameter factor for dependent likelihood function (standard deviation: likelihood_sd * y_error).
- likelihood_dependence (optional, required if likelihood_sd is present, default=False): To select whether to use the dependent likelihood function or the independent likelihood function.
- max_probability (optional, default=False): The acceptance probability will take the maximum probability value of the acceptance probability array if set true, otherwise the mean.

- iterations (optional, default=10000): Number of iterations.
- init_method (optional, default="random"): Specify the starting state of the Dream MCMC algorithm. Options: random, min, max, q1_prior, mean_prior, q3_prior, q1_posterior, median_posterior, q3_posterior.

Configurations for the parallel Metropolis-Hastings algorithm include:

- version (optional, default="ignoring"): Version of the MH algorithm. Options: ignoring, refl_bound, aggr.
- chains (optional, default=4): Number of chains.
- sd_transition_factor (optional, default=6): The standard deviation factor of the transition kernel. The standard deviation is given by (upper bound - lower bound) / sd_transition_factor.
- likelihood_sd (optional, default=1): The standard deviation parameter for independent likelihood function, or the standard deviation parameter factor for dependent likelihood function (standard deviation: likelihood_sd * y_error).
- likelihood_dependence (optional, required if likelihood_sd is present, default=False): Selects whether to use the dependent likelihood function or the independent likelihood function.
- max_probability (optional, default=False): The acceptance probability will take the maximum probability value of the acceptance probability array if set true, otherwise the mean.
- iterations (optional, default=2500): Number of iterations.
- init_method (optional, default="random"): Specify the starting state of the Dream MCMC algorithm. Options: random, min, max, q1_prior, mean_prior, q3_prior, q1_posterior, median_posterior, q3_posterior.

Configurations for the general parallel Metropolis-Hastings algorithm include:

- num_proposals (optional, default=8): The numbers of proposal points in each iteration.
- num_accepted (optional, default=4): The numbers of accepted points in each iteration.
- likelihood_sd (optional, default=1): The standard deviation parameter for independent likelihood function, or the standard deviation parameter factor for dependent likelihood function (standard deviation: likelihood_sd * y_error).
- likelihood_dependence (optional, required if likelihood_sd is present, default=False): Selects whether to use the dependent likelihood function or the independent likelihood function.
- sd_transition_factor (optional, default=6): The standard deviation factor of the transition kernel. The standard deviation is given by (upper bound - lower bound) / sd_transition_factor.
- version (optional, default="ignoring"): Version of the MH algorithm. Options: ignoring, refl_bound, aggr.
- iterations (optional, default=2500): Number of iterations.

- init_method (optional, default="random"): Specify the starting state of the Dream MCMC algorithm. Options: random, min, max, q1_prior, mean_prior, q3_prior, q1_posterior, median_posterior, q3_posterior.

Configurations for the DREAM algorithm include[2]:

- iterations (optional, default=1250): Number of iterations.
- chains (optional, default=8): Number of chains.
- DEpairs (optional, default=1): Number of chain pairs to use for crossover and selection of next point.
- multitry (optional, default=False): Whether to utilize multi-try sampling. It takes boolean or integer values.
- hardboundaries (optional, default=True): Whether to relect point back into bounds of hard prior.
- crossover_burnin (optional, default=0): Number of iterations to fit the crossover values.
- nCR (optional, default=3): Number of crossover values to sample from during run.
- snooker (optional, default=0): Probability of proposing a snooker update.
- p_gamma_unity (optional, default=0): Probability of proposing a point with gamma=unity.
- init_method (optional, default="random"): specify the starting state of the Dream MCMC algorithm. Options: random, min, max, q1_prior, mean_prior, q3_prior, q1_posterior, median_posterior, q3_posterior.
- likelihood_sd (optional, default=1): the standard deviation parameter for independent likelihood function, or the standard deviation parameter factor for dependent likelihood function (standard deviation: likelihood_sd * y_error).
- likelihood_dependence (optional, required if likelihood_sd is present, default=False): to select whether to use the dependent likelihood function or the independent likelihood function.

[2] More information regarding specifications can be found on `https://pydream.readthedocs.io/en/latest/pydream.html`

11. Conclusion and Further Outlook

In this thesis, the Bayesian inference of hydrological model parameters is implemented. Four versions of Markov chain Monte Carlo algorithms are used for performing Bayesian inference, each delivering different results based on their unique properties. The input algorithm parameter is then specified, in which the accuracy and efficiency metrics of different input algorithm parameters are benchmarked. For some input algorithm parameters, obvious relations between the configurations and the metrics can be found. For others, the configurations do not make a huge difference, or there is a certain configuration that stands out among all the different values of input algorithm parameters.

The first algorithm that is used is the fundamental Metropolis-Hastings, where one sample is generated in each iteration and later on, accepted or rejected based on the calculated acceptance probability using the likelihood function and sampling kernel. Therefore, the sampling kernel and the likelihood kernel play important roles, in which they exert a great impact on the acceptance or rejection.

The other three algorithms that are used all utilize the parallel aspect. The second algorithm, parallel Metropolis-Hastings is the parallel version of the fundamental Metropolis-Hastings algorithm, in which it uses multiple chains for sampling instead of one single chain. The number of chains is, therefore, a relevant factor for the result, since the number of samples generated in each chain is closely related to the convergence rate of the final result.

The third algorithm is the general parallel Metropolis-Hastings algorithm, in which multiple samples are generated in each iteration rather than one single sample. The acceptance probabilities are calculated in the form of a vector, which builds a probability space that allows random sampling to take place. Here, the ratio between the number of samples generated and accepted plays an indispensable role, as discussed in detail in the chapter. Other input algorithm parameters including sampling and likelihood kernels are also relevant, as they contribute to the calculation of the acceptance probability vector. In comparison to the two algorithms above, this algorithm achieves higher accuracy due to the acceptance mechanism, but potentially lower efficiency due to the complexity of acceptance calculation.

The final algorithm is the DREAM algorithm, which improves the sampling phase of the algorithm by adapting the sampling behavior based on past samples and employing a crossover mechanism from differential evolution to efficiently explore the parameter space. Thus, the configurations that are related to crossover, DE (differential evolution), and snooker are relevant for the performance of the result. The DREAM algorithm generally delivers the most accurate and efficient result, achieved not only by running multiple parallel chains but also by enhancing convergence and ensuring thorough exploration of the parameter space.

For the general implementation of Markov chain Monte Carlo algorithms, three other factors are relevant, namely the burn-in phase, the effective sample size, and the initial states. The burn-in phase discards the samples generated at the very start because the Markov chain has not entered the stationary phase yet. The effective sample size allows the algorithm to consider only every n-th sample in the result, allowing less dependency to form

between samples in the result that are next to each other. A good choice of initial states optimizes the efficiency of the sampling process by allowing the chains to quickly enter the phase where they sample from the stationary distribution.

For the specific use case for the hydrological model, handling the cases where the samples generated are out of bounds is also crucial. Since the prior parameters are uniformly distributed, the out-of-bounds samples could cause odd behaviors. Using mechanisms such as ignoring, reflection, and aggregation, these cases can be well handled.

Using the algorithms that are mentioned above, the parameters of the HBV-SASK model are inferred, but each to a different extent. While the fundamental and the parallel Metropolis-Hastings algorithms do not give out much information regarding the posterior, the general parallel Metropolis-Hastings and the DREAM algorithms present obvious results, both presented in the thesis. Both algorithms present similar posterior distributions for six of the seven dimensions. This shows the consistency of the inferred results for most cases, indicating the correctness of the inferred results. However, the samples generated by the DREAM algorithms are more concentrated in the posterior distribution, implying a better convergence of the generated samples. It is therefore the most optimal algorithm for the quantification of parameters under uncertainty. Descriptions of the posterior distribution of each dimension are listed below:

- TT: The distribution resembles a normal distribution with a peak around 2.8. The values are clustered around this mean, suggesting a small standard deviation.
- C0: The distribution resembles a normal distribution with a sharp peak around 2.1, with also a high concentration of values near the mean indicating a small standard deviation.
- β (beta): The distribution is roughly normal with its peak centered around 2.0. The values spread symmetrically from 1.5 to 2.5, indicating moderate variability.
- ETF: The distribution has a peak around 0.02 but with more spread, suggesting a normal distribution with a higher standard deviation.
- FC: The distribution shows a peak of around 250. It also has a narrow spread, which suggests a small standard deviation
- FRAC: The distribution appears to follow a normal distribution centered around 0.2. The spread from 0.15 to 0.25 indicates moderate variability.
- K2: The distribution exhibits a peak around 0.048, which is on the upper edge of the complete interval. Values spread narrowly on the left side of the peak.

Nevertheless, there is still plenty of work that can be done further regarding the topic of the thesis. For one, the acceptance rate of each algorithm could be further investigated. By tracking the percentage of iterations where the generated samples are accepted, the acceptance rate of the algorithms can be determined, so further improvement regarding sampling efficiency can be investigated and made. For another, the autocorrelation plots of generated samples using algorithms with different effective sample sizes can be examined. Effective sample size is a technique that is used to reduce the correlation between samples that are generated behind each other. The autocorrelation plots can be used to assess this aspect in a visual way, which is not conducted in this thesis. Besides, the DREAM algorithm has the potential to deliver more precise inferred results. Even though the DREAM

algorithm performs better than all other algorithms in terms of parameter quantification under uncertainty, the Bayesian inferred result does not exceed the accuracy of the one from the general parallel Metropolis-Hastings algorithm. Further tuning can be made around the parameters so that a more precise inferred result could potentially be derived. Apart from that, the anomaly of the testing data set is not well predicted. Even though we use this aspect to determine the configuration of the time series, on which the Markov chain Monte Carlo algorithms shall sample, the actual result of anomaly prediction is left to be desired. Finding a set of input algorithm parameter configurations and training data sets could be the key to a better result in terms of anomaly prediction. Last but not least, more algorithms of Markov chain Monte Carlo sampling can be tried for the Bayesian inference problem, so that algorithms with potential better performances in terms of the Bayesian inference problem for the hydrological model could be exploited and invented.

Part II.

Appendix

List of Figures

2.1. Metropolis-Hastings algorithm with symmetric distribution as transition kernel 7
2.2. Scaling of the proposal distribution in the Metropolis-Hastings algorithm . 8
2.3. Scaling of the proposal distribution 9
2.4. Posterior distribution derived from the Erlang distribution as proposal distribution 10

3.1. Time series decomposition of the Oldman dataset 15
3.2. Time series decomposition of the Banff dataset 16

5.1. Overview of the posterior distribution of the parameters calibrated by the default Metropolis-Hastings algorithm 26
5.2. Boxplots of the generated posterior samples of each parameter calibrated by the default Metropolis-Hastings algorithm 27
5.3. Comparison of Bayesian inference results of the default Metropolis-Hastings 28
5.4. Overview of the posterior distribution of the parameters calibrated by the Metropolis-Hastings algorithm that samples the bound value if the sample is out of bounds 30
5.5. Boxplots of the generated posterior samples of each parameter calibrated by the Metropolis-Hastings algorithm that samples the bound value if the sample is out of bounds 31
5.6. Overview of the posterior distribution of the parameters calibrated by the Metropolis-Hastings algorithm that reflect the samples into the inside of the range if the they are out of bounds 32
5.7. Boxplots of the parameters calibrated by the Metropolis-Hastings algorithm that reflect the samples into the inside of the range if the they are out of bounds 33
5.8. Comparison of the accuracy and the efficiency of Metropolis-Hastings algorithms based on the technique of handling samples generated out of bounds 34
5.9. Comparison of the accuracy and the efficiency of Metropolis-Hastings algorithms based on the sampling kernel standard deviation 36
5.10. Comparison of the accuracy and the efficiency of Metropolis-Hastings algorithms based on the likelihood function standard deviation 38
5.11. Comparison of the accuracy and the efficiency of Metropolis-Hastings algorithms based on the dependent likelihood function standard deviation . . . 39
5.12. Comparison of the accuracy and the efficiency of Metropolis-Hastings algorithms based on the implementation of probability acceptance probability calculation 41
5.13. Comparison of the accuracy and the efficiency of Metropolis-Hastings algorithms based on length of burn in phase 42

5.14. Comparison of the accuracy and the efficiency of Metropolis-Hastings algorithms based on the selection of effective sample size 44
5.15. Comparison of the accuracy and the efficiency of Metropolis-Hastings algorithms based on the number of iterations 45
5.16. Comparison of the accuracy and the efficiency of Metropolis-Hastings algorithms based on the selection of initial states 47
5.17. Overview of the posterior distribution of the parameters calibrated by the Metropolis-Hastings algorithm with tuned input algorithm parameters . . . 48
5.18. Boxplots of the generated posterior samples of each parameter calibrated by the Metropolis-Hastings algorithm with tuned input algorithm parameters . 49
5.19. Comparison of Bayesian inference results of the Metropolis-Hastings with tuned input algorithm parameters 50

6.1. Relationship between run time and chain numbers for parallel Metropolis-Hastings algorithm 53
6.2. Trace plot of the third chain from the parallel Metropolis-Hastings algorithm with 10 chains 55
6.3. Trace plot of the seventh chain from the parallel Metropolis-Hastings algorithm with 10 chains 56
6.4. Trace plot of the first chain from the parallel Metropolis-Hastings algorithm with 2 chains 57
6.5. Trace plot of the second chain from the parallel Metropolis-Hastings algorithm with 2 chains 58
6.6. Gelman Rubin Convergence Diagnostic of the parallel Metropolis-Hastings algorithm with 10 chains 60
6.7. Gelman Rubin Convergence Diagnostic of the parallel Metropolis-Hastings algorithm with 8 chains 61
6.8. Gelman Rubin Convergence Diagnostic of the parallel Metropolis-Hastings algorithm with 5 chains 62
6.9. Gelman Rubin Convergence Diagnostic of the parallel Metropolis-Hastings algorithm with 4 chains 63
6.10. Gelman Rubin Convergence Diagnostic of the parallel Metropolis-Hastings algorithm with 2 chains 64
6.11. Autocorrelation plot of the parallel Metropolis-Hastings algorithm with 10 chains 65
6.12. Autocorrelation plot of the parallel Metropolis-Hastings algorithm with 2 chains 66
6.13. Autocorrelation plot of the parallel Metropolis-Hastings algorithm with 8 chains 67
6.14. Mean RMSE of the parallel Metropolis-Hastings algorithm across test cases with different chains 68
6.15. Mean MAE of the parallel Metropolis-Hastings algorithm across test cases with different chains 69
6.16. Parameter overview by chain for parallel Metropolis-Hastings using 10 chains 71
6.17. Boxplot by chain for parallel Metropolis-Hastings using 10 chains 71
6.18. Parameter overview by chain for parallel Metropolis-Hastings using 2 chains 72
6.19. Boxplot by chain for parallel Metropolis-Hastings using 2 chains 72

7.1. Comparison of the accuracy and the efficiency of general parallel Metropolis-Hastings algorithms based on the ratio between numbers generated and accepted for each iteration . . . 75
7.2. Comparison of the accuracy and the efficiency of general parallel Metropolis-Hastings algorithms based on the amount of technique of handling samples generated out of bounds . . . 77
7.3. Comparison of the accuracy and the efficiency of general parallel Metropolis-Hastings algorithms based on the dependent likelihood function standard deviation . . . 78
7.4. Comparison of the accuracy and the efficiency of general parallel Metropolis-Hastings algorithms based on the selection of initial states . . . 80
7.5. Comparison of the accuracy and the efficiency of general parallel Metropolis-Hastings algorithms based on the dependent likelihood function standard deviation . . . 81
7.6. Overview of the posterior distribution of the parameters calibrated by the general parallel Metropolis-Hastings algorithm . . . 83
7.7. Boxplots of the generated posterior samples of each parameter calibrated by the general parallel Metropolis-Hastings algorithm . . . 84
7.8. Comparison of Bayesian inference results of the general parallel Metropolis-Hastings with tuned input algorithm parameters . . . 85

8.1. Relationship between run time and chain numbers for the DREAM algorithm 91
8.2. Trace plot of the third chain from the DREAM algorithm with 10 chains . . 92
8.3. Trace plot of the seventh chain from the DREAM algorithm with 10 chains 93
8.4. Trace plot of the first chain from the DREAM algorithm with 4 chains . . . 94
8.5. Trace plot of the third chain from the DREAM algorithm with 4 chains . . 95
8.6. Gelman Rubin Convergence Diagnostic of the DREAM algorithm with 10 chains . . . 96
8.7. Gelman Rubin Convergence Diagnostic of the DREAM algorithm with 8 chains 97
8.8. Gelman Rubin Convergence Diagnostic of the DREAM algorithm with 5 chains 98
8.9. Gelman Rubin Convergence Diagnostic of the DREAM algorithm with 4 chains 99
8.10. Autocorrelation plot of the DREAM algorithm with 10 chains . . . 100
8.11. Autocorrelation plot of the DREAM algorithm with 8 chains . . . 101
8.12. Autocorrelation plot of the DREAM algorithm with 5 chains . . . 102
8.13. Autocorrelation plot of the DREAM algorithm with 4 chains . . . 103
8.14. Mean RMSE of the DREAM algorithm across test cases with different chains 104
8.15. Mean MAE of the DREAM algorithm across test cases with different chains 105
8.16. Parameter overview by chain for DREAM using 10 chains . . . 106
8.17. Boxplot by chain for DREAM using 10 chains . . . 107
8.18. Parameter overview by chain for DREAM using 4 chains . . . 108
8.19. Boxplot by chain for DREAM using 4 chains . . . 108
8.20. Comparison of the accuracy and the efficiency of DREAM algorithms based on the HardBoundaries parameter . . . 110
8.21. Comparison of the accuracy and the efficiency of DREAM algorithms based on the crossover burn in parameter . . . 111

8.22. Comparison of the accuracy and the efficiency of DREAM algorithms based on the adaptive crossover parameter . . . 112
8.23. Comparison of the accuracy and the efficiency of DREAM algorithms based on the nCR parameter . . . 113
8.24. Comparison of the accuracy and the efficiency of DREAM algorithms based on the independent likelihood kernel factor . . . 115
8.25. Comparison of the accuracy and the efficiency of DREAM algorithms based on the dependent likelihood kernel factor . . . 116
8.26. Comparison of the accuracy and the efficiency of DREAM algorithms based on the initialization method . . . 117
8.27. Comparison of the accuracy and the efficiency of DREAM algorithms based on the p gamma unity parameter . . . 118
8.28. Comparison of the accuracy and the efficiency of DREAM algorithms based on the DEpairs parameter . . . 120
8.29. Comparison of the accuracy and the efficiency of DREAM algorithms based on the snooker parameter . . . 121
8.30. Comparison of the accuracy and the efficiency of DREAM algorithms based on the burn in factor . . . 123
8.31. Comparison of the accuracy and the efficiency of DREAM algorithms based on the effective sample size . . . 124
8.32. Overview of the posterior distribution of the parameters calibrated by the general parallel Metropolis-Hastings algorithm . . . 125
8.33. Boxplots of the generated posterior samples of each parameter calibrated by the general parallel Metropolis-Hastings algorithm . . . 126

9.1. The Bayesian inferred result of the posterior sampled using the short and calm time series, testing on data containing floods . . . 129
9.2. The Bayesian inferred result of the posterior sampled using the short with peaks time series, testing on data containing floods . . . 130
9.3. The Bayesian inferred result of the posterior sampled using the long and calm time series, testing on data containing floods . . . 131
9.4. The Bayesian inferred result of the posterior sampled using the long with peaks time series, testing on data containing floods . . . 132
9.5. The Bayesian inferred result of the posterior sampled using the short and calm time series, testing on data displaying calmness. All of the test cases on data displaying calmness share similar results to this visualization . . . 133
9.6. Comparison of metrics for the test case of short periods . . . 134
9.7. Comparison of metrics for the test case of long periods . . . 135
9.8. Comparison of metrics for the test case of periods with peaks . . . 136
9.9. Comparison of metrics for the test case of periods from different basins . . . 137
9.10. Comparison of metrics for the observation of spin-up length . . . 138