ShinyHastings: An Illustration of the Effect of the Proposal Type on the Metropolis Hastings Algorithm

Zarah Leonie Weiß
zarah-leonie.weiss@student.uni-tuebingen.de

Abstract

This article introduces *ShinyHastings*, a Shiny app web application that allows users to experience the effect of choosing varying proposal types for the approximation of a probability distribution. It implements five different proposal types, of which three belong to the family of random-walk Metropolis Hastings (MH) algorithms and two to the family of independence MH algorithms. In the application, they can be used to approximate mean and standard deviation of a set of different target distributions, symmetric as well as asymmetric ones.

1 Introduction

Markov chain Monte Carlo (MCMC) algorithms are important tools for the approximation of probability distributions and densities. Unlike other sampling methods, such as for example grid searches, MCMC methods allow to approximate computationally costly probability distributions, for example multi-dimensional ones with a large amount of parameters (Hastings 1970, p. 97, Kruschke 2015, p. 144) or non-standard distributions, that have not been neatly described by statistics (Jackman 2009).

MCMC methods are particularly beneficial for Bayesian inference approaches, where the posterior probability has to be calculated. How so? While a proper introduction to Bayesian statistics would be beyond the scope of this article[1], it seems worthwhile to recapitulate how Bayesian inference imposes computational challenges that can be conquered using MCMC methods.[2] Bayes' theorem is given in Equation 1.

$$P(\theta|D) = \frac{P(\theta)P(D|\theta)}{\int P(\theta^*)P(D|\theta^*)d\theta^*} \tag{1}$$

It shows that the posterior probability $P(\theta|D)$ can be calculated by normalizing the product of the prior probability $P(\theta)$ and the likelihood $P(D|\theta)$ with the marginal likelihood $\int P(\theta^*)P(D|\theta^*)d\theta^*$. The computation of the marginal likelihood is computationally very costly, though, unless θ is a single discrete variable with a restricted domain size and the prior distribution is a conjugate prior for the likelihood function.[3] Also, the resulting posterior distribution may very well turn out to be a non-standard distribution, from which we cannot directly sample (ibid., p. 153). However, using MCMC methods, the denominator may be dropped: for MCMC algorithms it is sufficient to sample from distributions proportional to a target distribution instead of using the actual distribution. Therefore, it suffices to sample from the product of prior and likelihood (see Equation 2) in the case of approximating the posterior.

$$P(\theta|D) \propto P(\theta)P(D|\theta) \tag{2}$$

While MCMC methods are very powerful in the sense that they allow us to approximate distributions we could not approximate otherwise, they have

[1] This would also be superfluous considering the vast amount of introductions to the topic, see for example Jackman (2009), Kruschke (2015), Robert (2016), and Robert & Castella (2010).

[2] It also gives me a convenient excuse to cite Bayes theorem in my introduction, which seems to be constitutive for all contributions even remotely related to Bayes.

[3] See `http://www.sfs.uni-tuebingen.de/~mfranke/bda+cm2015/slides/05_MCMC`, page 3.

the downside of being computationally costly themselves. On the one hand, all sampling methods share this characteristic in the sense that a large amount of samples is needed to approximate a parameter. On the other hand, MCMC methods are less efficient and require more iterations than, for example, simple Monte Carlo methods, because they do not draw truly random samples from a distribution: as the term 'Markov chain' being part of the name suggests, the samples are dependent on each other, which makes the algorithms less efficient (Jackman 2009, p. 172).

Therefore, it is important to tune MCMC algorithms' efficiency when implementing them, if possible. The Shiny app[4], which this article is introducing, was designed to facilitate the awareness of how to do so for a common MCMC method: the MH algorithm. The efficiency of MH is heavily influenced by the choice of *acceptance ratio* and *proposal distribution*. The *ShinyHastings* app focusses on the latter and allows users to immediately experience the effect of different proposal distribution implementations on the MH algorithm. This article introduces *ShinyHastings* and also briefly reflects the benefits and shortcomings of using Shiny as interface framework.

The remainder of the article is structured as follows: in section 2 a comprehensive overview over the theoretical background of MCMC methods in general (subsection 2.1) and the MH algorithm in particular (subsection 2.2) is given. Afterwards, *ShinyHastings* is introduced in section 3, which briefly introduces the interface and implementional details as well as some criticism on Shiny, before discussing the different proposal distributions available. The article closes with the conclusion in section 4.

2 Theoretical Background

2.1 Markov Chain Monte Carlo Methods

MCMC algorithms may be used to approximate virtually any *target* distribution via sampling, even if direct sampling from that target distribution is computationally not feasible (Liu 2001, p. 105, Kruschke 2015, p. 144). This is due to the fact that it suffices for the algorithms to sample from distributions that are proportional to the target distribution (Jackman 2009, p. 140).[5] The target distribution is then characterized based on summary statistics of those samples, such as mean, central tendency estimates or Highest Density Interval (HDI) (Kruschke 2015, p. 145).

While MCMC methods may sample from a distribution proportional to the target distribution, it is crucial that such a proportional distribution can be computed. Applying MCMC to Bayesian inference methods, this means that prior and likelihood function need to be computable easily for a given parameter θ. Based on this, MCMC returns an approximation of $P(\theta|D)$ in form of a large set of samples of θ values (ibid., p. 144).

In order to understand how MCMC methods operate, it is necessary to elaborate on the two basic underlying components, which name this set of algorithms: the *Monte Carlo principle* and the *Markov Chain property*.

Monte Carlo Principle Monte Carlo methods are used to approximate distributions by sampling, just as MCMC methods, which are in fact – as the name suggests – extensions of Monte Carlo methods. However, standard Monte Carlo methods are based on truly random, i.e. independent, samples, that are actually drawn from the target distribution. They have been employed to solve deterministic and statistical problems long before automatic computation as we know it was possible; Jackman (2009, p. 140) even refers to a paper from 1873.

The concept of drawing conclusions about a population based on sampling is not new; in fact, it is applied in experimental set-ups all the time. However, applying Monte Carlo methods we sample from mathematically defined distributions, which is conceptually quite different from experimental setups (cf. Kruschke 2015, p. 145). Underlying both approaches is the Law of Large Numbers (LLN), which states that the average value in a sequence

[4] `http://shiny.rstudio.com`.

[5] Another method of distribution approximation without directly sampling from the actual target distribution is *importance sampling*, where samples instead are drawn from a *trial* distribution resembling the actual target distribution. However, it would be beyond the scope of this article to discuss this alternative approach, please see Hastings (1970), Jackman (2009), and Liu (2001) instead.

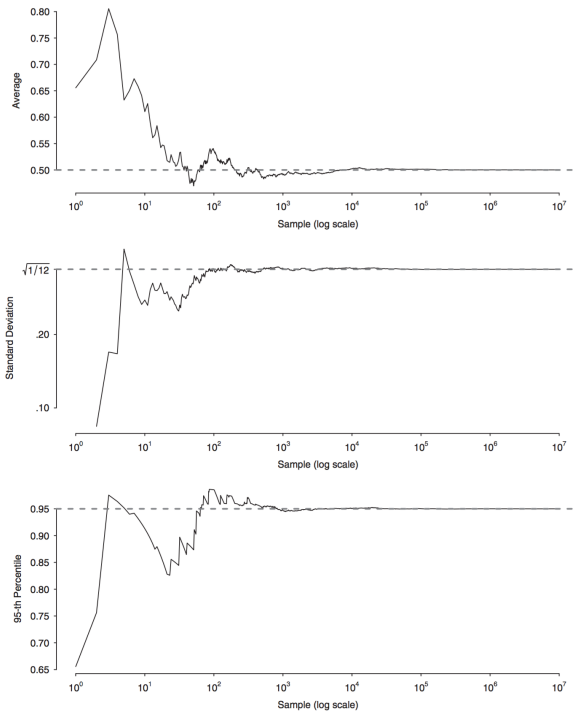


Figure 1: Statistical summary of Monte Carlo sampling history for $\theta \sim \mathcal{U}(0, 1)$, cf. Figure 3.1 in Jackman (2009, p. 136).

of trials will approach the expected value as the sequence grows longer. Based on this, the Monte Carlo principle states, that the statistical summary of a large enough amount of samples from a density $f(\theta)$ qualifies as an estimate of a property of θ, i.e. it is *simulation consistent* and can be used as an approximation of the actual density (cf. Jackman 2009, 133f, Robert 2016, p. 1). For a formal definition of simulation consistency please consult Jackman (2009, p. 138). Figure 1 illustrates this for mean, standard deviation and the 95th percentile, displaying the Monte Carlo sampling history for $\theta \sim \mathcal{U}(0, 1)$. Unlike numerical methods, Monte Carlo methods are especially efficient for sampling from multi-dimensional distributions (cf. Hastings 1970, p. 97). However, as Figure 1 also illustrates, it takes a large amount of samples in order to achieve a proper approximation. For many cases this is not a computational problem any more, though.

Markov Chain Property While the Monte Carlo principle plays a crucial part in MCMC methods, it does not allow to approximate distributions from

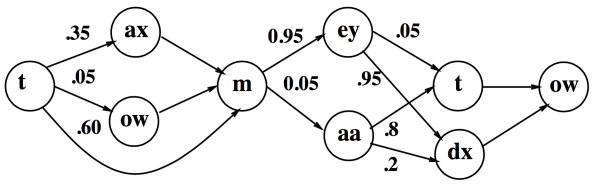


Figure 2: A Markov chain illustrating the pronounciation of the word *tomato*, cf. Figure 5.12 in Jurafsky & Martin (1999, p. 168).

which direct random sampling is not possible. This is, where the Markov chain property comes into play (Robert 2016, p. 1): it imposes only minimal requirements on the distribution to sample from, which is good, if we do not know the distribution as is the case for Bayesian posteriors, and it allows to decompose high dimensional problems into smaller parts (Robert & Castella 2010, p. 168). This paragraph gives a brief introduction to Markov chains, just as detailed as necessary for the purposes of discussing the MH algorithm. Then it discusses how Markov chains and the Monte Carlo principle can be combined to MCMC methods.

The Markov chain property was proposed by the Russian mathematician Andrey Markov (1856-1922) and defines a special form of a weighted automaton, in which the iteration history of the automaton is uniquely defined by the input sequence.[6] An example Markov chain for the pronunciation of *tomato* is displayed in Figure 2. A Markov chain is a stochastic process and can be thought of as a sequence of random variables $\{X^{(t)}\} = X^{(1)}, \ldots, X^{(t)}$, if for all t the Markov chain property displayed in Equation 3 holds.

$$\begin{aligned} &P(X^{(t+a)} = y \mid X^{(s)} = X_s, s \leq t) \\ &= P(X^{(t+a)} = y \mid X^{(t)} = x_t), \forall a > 0, \end{aligned} \tag{3}$$

It states, that the probability distribution of $X^{(t+a)}$, i.e. its transition kernel K, is dependent only on the immediately preceding a random variables or *states* of the chain (Jurafsky & Martin 1999, p. 167), i.e. $K(X^{(t+a)}, X^{(t)})$ (Robert & Castella 2010, p. 168).

[6] Weighted automatons are regular finite state automatons in which the arcs between states are associated with transition probabilities. For more details on automata, please see Jurafsky & Martin (1999).

In the case of first-order Markov models $a = 1$, i.e. the distribution of the current state $X^{(t+1)}$ is dependent only on the preceding state $X^{(t)}$. Equation 3 can, hence, be reduced to Equation 4.

$$\begin{aligned} P(q^{(t+1)} = y \mid q^{(s)} = x_s, s \leq t) \\ = P(q^{(t+1)} = y \mid q^{(t)} = x_t) \end{aligned} \tag{4}$$

In the following, we are discussing first-order Markov chains. However, all claims may easily extended to higher order Markiv chains as well.

How does this combine with Monte Carlo methods? We may construct a Markov chain on the parameter space Θ, i.e. use the parameter space of the posterior distribution as the state space of our Markov chain. In classical Markov processes, we try to determine such a stationary distribution given the transition rule for the Markov chain. MCMC simulations, though, are constructed such that they use a distribution π as their stationary distribution and try to predict an efficient transition rule (cf. Liu 2001, p. 106).

A Markov chain has a stationary distribution $\pi(X)$, if Equation 5 holds:

$$X^{(t)} \sim \pi, X^{(t+1)} \sim \pi, \forall X^{(t)} \in \Theta, \tag{5}$$

i.e. a Markov chain has a stationary distribution, if for all possible states in this Markov chain it holds, that they are drawn from the same distribution as their preceding state. In order to ensure a Markov chain has such a unique stationary distribution, it must be positive recurrent, aperiodic and irreducible (see e.g. Metropolis et al. 1953; Robert 2016).

Positive recurrent A Markov chain is said to be *positive recurrent*, if for any sequence of states it is possible to return to this sequence in a finite number of steps.

Aperiodicity *Aperiodicity* means, that this finite number of steps is not necessarily a multiple of come constant c.

Irreducibility *Irreducibility* means, that the transition kernel K allows to reach every state in Θ from every other state at least by mediation of other states.

Ergodicity Irreducible Markov chains are said to be *ergodic*: through iterations within irreducible Markov chains the potential values of θ in the parameter space are visited proportional to the probability assigned to them by the stationary distribution (Jackman 2009, p. 172). Therefore, the iteration sequences of those Markov chains constitute representative samples of their target densities by the Monte Carlo principle, although these samples are not independent from each other. In fact, *ergodictiy* is a form of the LLN (ibid., p. 171), hence MCMC Markov chains successfully model probability densities proportional to the target densities we want to approximate.

To summarize: MCMC algorithms approximate distributions simulating an irreducible, positive recurrent, aperiodic Markov chain whose stationary distribution is at least proportional to the target distribution and they use the iteration history as samples, resulting in a transition rule for the target distribution (Liu 2001, p. 106, Jackman 2009, p. 201).

As already briefly mentioned, the correlation between samples comes at some costs: the estimation of the standard deviation and error for an estimate become more difficult in these cases, since the sample variance is artificially increased compared to independent samples. Hence, more iterations are necessary to achieve representable results (cf. Hastings 1970, p. 98, Liu 2001, p. 106). For practical purposes this may lead to implementations where convergence cannot be achieved in a reasonable time. Also, in Bayesian settings MCMC might not converge if the product of prior and likelihood is not integrable (Robert & Castella 2010, p. 170). It is, therefore, crucial how to implement MCMC algorithms, such as the MH algorithm, which is introduced in the following section.

2.2 Metropolis Hastings Algorithm

MH is a well-known core MCMC algorithm (Jackman 2009, p. 201), which is applied in a broad range of areas, such as biology, chemistry, computer science, economics, engineering, material science, physics and statistics (cf. Liu 2001, p. 106). It was first introduced by Metropolis et al. (1953) and later on generalized by Hastings (1970).

The algorithm is similar to simple two-fold acceptance-rejection sampling as it consists of a proposal and a decision phase in each iteration step. After randomly choosing the initial state $\theta^{(0)}$ from the state space Θ. The algorithm determines in which state to be for each iteration step $t > 0$ as follows:

Proposal phase First, propose a new random variable θ^* different from the current θ^t based on the proposal distribution $J(\theta^*, \theta^{(t)})$.

Decision phase Second, the next state $\theta^{(t+1)} = \theta^*$, if $P(\theta^{(t)}) > P(\theta^*)$, i.e. the proposed state becomes the next state, if the proposed state is more likely than the current state. Else, calculate the movement probability $\alpha = min(1, r(\theta^{(t)}, \theta^*))$. Draw a random sample from a uniform distribution in the interval $[0, 1]$. If $u \leq \alpha$, still move to θ^* probabilistically, i.e. still $\theta^{(t+1)} = \theta^*$. Else, stay at $\theta^{(t)}$, i.e. $\theta^{(t+1)} = \theta^{(t)}$.[7]

Obviously, it is crucial for the algorithm how acceptance ratio r and proposal distribution J are defined. The original Metropolis algorithm by Metropolis et al. (1953) defined r as shown in Equation 6.

$$r(\theta^{(t)}, \theta^*) = \frac{P(\theta^*)}{P(\theta^{(t)})} \tag{6}$$

However, this acceptance ratio assumes a symmetric proposal distribution, i.e. $J(\theta^*, \theta^{(t)}) = J(\theta^{(t)}, \theta^*)$. Clearly, this limits the distributions that can efficiently be approximate with the algorithm: if the target distribution is asymmetric, an asymmetric proposal distribution would be a preferable choice. Therefore, Hastings (1970) introduced a generalized version of the algorithm, known was the MH algorithm, using the acceptance ratio displayed in Equation 7, which also allows for asymmetric proposal distributions.

$$r(\theta^{(t)}, \theta^*) = \frac{P(\theta^*)}{P(\theta^{(t)})} * \frac{J(\theta^*, \theta^{(t)})}{J(\theta^{(t)}, \theta^*)} \tag{7}$$

Clearly, for symmetric distributions the second multiplier reduces to 1, resulting in Equation 6. There are other definitions of the acceptance ratio, however, they will not be discussed in further detail. For an overview, please see Liu (2001, 111f).

The optimal choice of the proposal distribution J is also broadly discussed in literature, see e.g. Jackman (2009) and Liu (2001). In the following, two commonly used lines of proposal types are discussed: *random-walk* proposals and *independence* proposals. In random-walk metropolis proposals $X^{(t+1)}$ is a random perturbation ϵ of the previous state $X^{(t)}$ (Robert & Castella 2010, p. 182), which is formalised in equation 8.

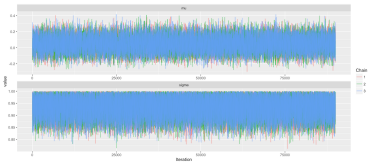
$$X^{(t+1)} = X^{(t)} + \epsilon_t, \epsilon_t \sim g(\cdot), \tag{8}$$

with $g(\cdot)$ being some spherically symmetric distribution (Liu 2001, p. 114). MH algorithms using such proposals build so called random-walk Markov chains, if J is symmetric. They are, therefore, called random-walk MH algorithms (Robert & Castella 2010, pp. 169,182) (also in the non-symmetric case, where they are, strictly speaking, no random walks anymore). The efficiency of these algorithms is heavily influenced by the amount of perturbation from the current state: If it is too small, the sample space is not explored sufficiently. For too large perturbations, most proposals will be rejected, though, also resulting in an inefficient proposal distribution (Liu 2001, p. 118). In both cases convergence time and auto-variance increase (Robert & Castella 2010, p. 183). Figure 3 illustrates this clearly with the trace plots for the Simple Uniform Random-Walk MH in *ShinyHastings* (see section 3.3), where the perturbation $\epsilon_t \sim \mathcal{U}(-\gamma, \gamma)$ is set with $\gamma = [.0001, .2, 10]$, otherwise using the *ShinyHastings* default settings.

While *random-walk* proposals are dependent on the current state $\theta^{(t)}$, *independence* proposals are sampled independently from $\theta^{(t)}$, for example by using the prior or the likelihood as proposal distribution. This makes the algorithm more efficient and decreases auto-variance (Jackman 2009, 212ff). For both proposal types holds, that, albeit inefficient, from a formal point of view it is not crucial to ensure that the proposals lie within the target distributions as they will be rejected most of the time (Robert & Castella 2010, p. 185).

[7] This differs from common acceptance-rejection algorithms, in which the final rejection of the proposed state leads to a new current state randomly sampled from the proposal distribution (cf. Chib & Greenberg 1995, p. 329).

(a) $\gamma = .0001$



(b) $\gamma = .2$

(c) $\gamma = 10$

Figure 3: Trace plots for default sampling from a normal distribution with $\mu = 0, sd = 1$ in ShinyHastings using a Simple Uniform Random-Walk MH with varying γ values.

3 ShinyHastings App

ShinyHastings is a web application implemented in Shiny[8], a web application framework for R developed by RStudio[9]. It can be accessed via GitHub[10] at `https://github.com/zweiss/ShinyHastings`. To call it directly via *R*, the following command can be used:

```
runGitHub("zweiss/ShinyHastings")
```

The app was designed to allow to run the MH algorithm using varying proposal distributions J, thereby illustrating their effect on the algorithms efficiency given different types of target distributions. It is called *ShinyHastings*, because, unlike most implementations of the MH algorithm that can be found at the internet, it does not implement the original Metropolis algorithm, which is limited to symmetric proposal distributions, as pointed out in section 2.2. Instead, it uses the generalised acceptance ratio by Hastings (1970) in order to allow for asymmetric proposal distributions, too.

[8] `http://shiny.rstudio.com`.

[9] `https://www.rstudio.com`.

[10] `https://github.com`.

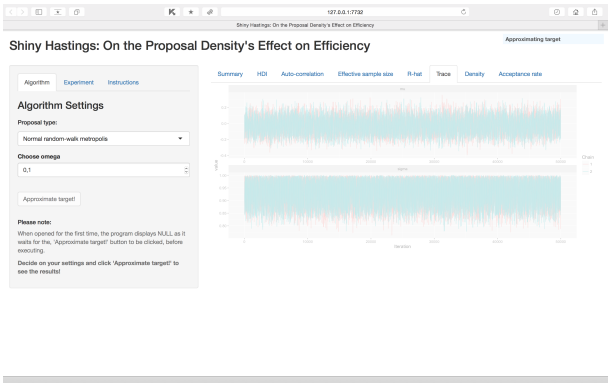


Figure 4: The *ShinyHastings* interface.

In the following, first the user interface is briefly introduced in section 3.1, before discussing the server logic and especially the different types of proposal densities implemented in section 3.3.

3.1 User Interface

The interface is designed with a sidebar layout separating the settings panel on the left or, depending on the browser size, top of the application from the result panel at the right or bottom. This is illustrated in figure 4.

3.1.1 Settings Panel

As can be seen, the settings panel consist of three tabs and allows users to set-up their individual experiments in terms of proposal type, target distribution, sampling duration, etc. The tabs are structured as follows:

Algorithm settings The algorithm settings allow to select a proposal type and some parameters that are dependent on the proposal type. Changes on the settings are not immediately executed. Instead, after all changes have been made, the user may initiate a new sampling process by clicking the 'Approximate target!' action button. An ongoing computation is then indicated to the user in the right corner of the main panel, also, the action button is disabled during runtime.

Experimental settings In the experimental settings the user may choose the target distribution, which should be approximated in terms of distribution type, mean, and standard deviation. It can be

chosen between a symmetric normal distribution and an asymmetric log normal distribution, to simulate differences between MH algroithms. It also allows to adjust the initial parameters for the MH algorithm. This should not effect the performance in a long run, however, as the initial state becomes unimportant in a Markov chain. Yet, the idea was to give the user as much freedom as possible. Also, the iteration set-up may be chosen in terms of chain, iteration and burn-in number.[11]

Instructions The instruction tab displays a short instruction on how to use the program, also pointing to this article and a more elaborate readme file.

3.1.2 Result Panel

The result panel outputs the results of a sampling process in terms of a set of summary statistics, which allow the user to evaluate the performance of their sampling. It is based on the R packages CODA[12] and GGMCMC[13] and features the following tabs:

Summary The summary tab displays the summary statistics from the posterior distribution, i.e. mean values that were approximated for the variables as well as the standard deviation throughout the iterations and their quantiles. It is generated calling the generic `summary()` function on the MH output in form of a CODA mcmc.list object.

HDI The HDI tab displays the 95% highest density intervals for the variables, i.e. the interval in which 95% of the distribution lies. It is generated using the `ggs_autocorrelation()` function from the GGMCMC package.

Trace The trace tab allows for visual inspection of the sampling result. It plots the sample values (y-axis) for each iteration step (x-axis) for each chain (encoded in colors). Ideally, the chains are heavily overlapping. The trace plot is generated using the GGMCMC `ggs_traceplot()` function.

Density The density tab displays the approximated posterior density for each chain. Ideally, those resemble the target distribution and are overlapping. The density is plotted with the GGMCMC `ggs_density()` function.

Auto-correlation Auto-correlation plots the auto-correlation of the samples across time lags, with ± 1 indicating perfect negative or positive correlation. In MCMC algorithms auto-correlation indicated how much the samples depend on each other. As already discussed in section 2.1 dependence makes the algorithm inefficient. Therefore, it is desirable for auto-correlation to drop against 0 for $lag \geq 1$. It is generated using the `ggs_autocorrelation()` function from the GGMCMC package.

Effective sample size The effective sample size tab shows the sample size adjusted for auto-correlation: N dependent samples are equivalent to $N_{effective}$ independent samples in terms of credible information on the distribution from which was sampled. Ideally, the effective sample size is high, meaning the algorithm performs relatively efficient. In order to display the effective sample size, the `effectiveSize()` function from the CODA package is called.

R-hat The Potential Scale Reduction Factor, in short $\hat{R}$ factor, is displayed in the R-hat tab. It was proposed by Gelman & Rubin (1992) in order to diagnose convergence in MCMC methods by comparing within and between chain variance. If $\hat{R} > 1$, the algorithm did not converge. The results in this tab are obtained by calling the `gelman.diag()` function from the CODA package.

Acceptance rate The acceptance rate is the ratio of accepted proposals to made proposals in the chains, excluding the burn-in phase. A good acceptance rate is suggested to be between 25% to 35% (Liu 2001, p. 115). It is calculated using the

[11] The input form for the number of iterations and burn-ins was chosen to be a slider, because the number of burn-ins is in fact subtracted from the number of iterations and cannot exceed it. This could be ensured and visualised by this choice. This comes at the cost of limiting the maximal number of iterations to 100,000 in order to get a manageable slider. However, as larger numbers or iterations also extend run-time and the implemented algorithms can achieve reasonable results for between 10,000 and 100,000 iterations, this was considered to be preferable over the alternative input measures.

[12] `https://cran.r-project.org/web/packages/coda`.

[13] `https://cran.r-project.org/web/packages/ggmcmc`.

CODA `rejectionRate()` function and subtracting it from 1, i.e. $1 - rejectionRate()$.

Distribution selection Unlike the other tabs in the main panel, *distribution selection* does not display any result statistics. Instead, it plots the chosen target distribution, in order to allow the user to visually inspect the settings with regard to the distribution choices.

Used settings This tab, too does not display some sort of statistics, but records the settings used for the approximation. It is designed to help users recall which configuration the displayed results come from, even after altering the settings in the settings panel.

3.2 Server Side

Before discussing the implemented proposal types in greater detail, a few words about some implementation aspects should be made.

Runtime Handling Approximating the target distribution may take a while, depending on machine capacity and iteration set-up. In a web application this is especially troublesome for users. Hence, several arrangements were made when developing the application in order to ensure a satisfying user experience:

Contrary to the default in Shiny applications, the application was designed to be not fully executed when starting *ShinyHastings*, insofar as it does not start sampling using the pre-set settings. Instead, it waits for the user to initiate the sampling process by clicking the action button at the algorithm settings tab. This also holds for adjustments made in the settings panel. The application waits for all changes to be made, indicated by clicking the action button, before re-executing.

Also, while the application approximates the given target distribution, the action button is disabled. This prevents the user from accidentally starting an increasing number of sampling processes, which would be executed one after another. For disabling the action button, the SCHINYJS package by Dean Attali was used.[14] Additionally, a progress message is displayed in the right upper corner of the result panel, in order to inform the used that the program is in fact still running. New changes to the settings may be made during runtime, however, in order to prepare the next sampling experiment.

Troubleshooting Part of discussing the server side of *ShinyHastings* should also be a brief review of the framework used to design the interface. While Shiny overall is a great resource when it comes to equip R code with a visually appealing and easily set-up interface, in the course of designing the application some issues surfaced, which should be mentioned.

The first minor flaw is that the conditional panel seems to break down if conditions are nested too deeply. *ShinyHastings* employs conditions in order to hide additional settings in the experiment settings tab and in order to differentiate between the tuning parameters of the different proposal types. However, I found that this functionality breaks down for increasingly depths of nesting, leading to a display of all widgets in the condition panel whether a condition is met or not. Consistent with this observation is, that at some point radio buttons were displayed but became frozen when linking them to an elaborate condition and displaying them in a row with overall three widgets, which itself was linked to a conditional panel. However, these issues occurred while testing elaborate structures, which were meant to test what could be done extensively using conditional panels and columns. These issues, therefore, seem negligible.

A second issue became apparent when experimenting with the DISTR package: some R packages might overwrite native Shiny methods, leading to errors after quitting and re-starting the application within the same R session. The following puzzling error occurred when using the DISTR package with Shiny: the application did not start properly any more every time the application was run for the second time via RStudio, displaying the following error:

> ERROR: unable to find an inherited method for function ‘p’ for signature ‘"character"’

In order to overcome it, RStudio needed to be shut down and restarted completely. A termination of the

[14] For more information, please see `http://deanattali.com/2015/04/23/shinyjs-r-package/`.

current R session was not sufficient. As the `p()` function is a package native to Shiny, this seems puzzling at first. What has happened? DISTR, too, uses a `p()` function, obviously with a different functionality, though. Therefore, both packages are not compatible, because DISTR masks the Shiny function. Still, the program executes properly in the first run, as it first executes the user interface, which uses `p()` and then the packages from from the helper file. However, when executing it the second time without restarting R, the application fails to load the user interface, as `p()` has been masked. While probably not severe in most cases, when using Shiny one should keep this potential issue in mind.

Algorithm Implementation There are many ways to implement the general MH algorithm as it was outlined in section 2.2. Therefore, it seems worthwhile to briefly discuss its actual implementation in *ShinyHastings*. As six different sub-types of the MH algorithm have been implemented, which differ in the type of proposal and acceptance probability, but are otherwise identical, a single `metropolis_hastings()` function was defined[15], which always calls the proposal and acceptance function, that have been passed as parameters. In order to translate the information from the settings chosen by the user, that are passed on by the Shiny front end, into the correct parameters for the general `metropolis_hastings()` function, a wrapper function is employed. Other than that, for each possible proposal function J two functions are defined: the proposal function and its acceptance function. The code also contains log posterior, log prior and log likelihood functions.[16] Pseudo code for the `metropolis_hastings()` function is displayed below.

```
for (c in 1:chains) {

 # I. Initialization phase

 # start with random samples
 mu ~uniform(muMin,muMax)
 sigma ~uniform(sigmaMin,sigmaMax)
 plog_cur=posterior(mu,sigma)

  for (i in 1:iterations) {

   # II. Proposal phase

   # get proposals
   muNext
    ~proposal((mu),tuning_param)
   sigmaNext
    ~proposal((sigma),tuning_param)

   # get probabilities for proposals
   # (use log probabilities)
   plog_next
    =posterior(muNext,sigmaNext)
   rlog
    =rlog(muNext,sigmaNext,mu,sigma)

   # III. Decision phase

   # acceptance probabilities
   p_accept
    =exp(plog_next-plog_cur+r_log)

   rndm~uniform(0,1)
   if(rndm<p_accept) {
    mu=muNext
    sigma=sigmaNext
    plog_cur=plog_next
  }
 }
}
```

As can be seen, for each chain the mean and variance are initialized randomly within the interval set by the user in the advanced settings. Also, an initial log posterior probability is defined. Then, for each iteration proposal and decision phase from the MH algorithm are performed. In the proposal phase new values for mean and standard deviation are drawn from the proposal function `proposal()`, which differs based on the user's selection. The current state is given as an optional parameter, as it is not required in independence MH types. The tuning parameters differ between proposal functions and are also set by the user at the algorithm settings tab. Then, the

[15] Using the R packages CODA and GGMCMC.

[16] Log values were calculated in order to avoid numerical underflow caused by very small likelihood values. The log acceptance ratio is transformed to the actual likelihood when calculating the acceptance probability.

log posterior probability for the proposed parameter is calculated as well as the acceptance ratio, which also differs based on the choice of proposal function. In the following decision phase, the acceptance probability is calculated. Note, that this is equivalent to Equation 7, since the code uses logarithms. Then, the proposal is probabilistically accepted. The implemented version is reduced but equivalent compared to what was introduced in section 2.2: it is superfluous to calculate $\alpha = min(p\_accept, 1)$, as the condition in the if statement is always true for $p\_accept \geq 1$, even though it is not a proper probability for values exceeding 1. If the probability to accept is less than 1, the proposal is accepted probabilistically, which is realised in form of the random variable drawn from a uniform distribution between 0 and 1.[17]

3.3 Proposal Types

ShinyHastings features five different proposal types: three types of random-walks and two types of independence samplers. This section gives a brief introduction to each of them, discussing some theoretical background as well as implementation details.

3.3.1 Random-Walk Metropolis Hastings

Simple Uniform Random-Walk MH The most popular type of proposal distribution is a simple random-walk MH sampling using a uniform distribution for $g(\cdot)$ (Liu 2001, p. 114, Jackman 2009, p. 204), as illustrated in Equation 9.

$$\begin{aligned} X^* &= X^{(t)} + \epsilon_t, \\ \epsilon_t &\sim \mathcal{U}(-\gamma, \gamma) \end{aligned} \tag{9}$$

with γ being the tuning parameter. As already illustrated in figure 3 in section 2.2, the choice of ϵ and, therefore, of γ, is crucial for the efficiency of the algorithm. Therefore, in *ShinyHastings*, this γ can be manipulated by the user in the *Algorithm* tab at the *Settings* panel.

[17] Plase note that this code is based on an implementation by Fabian Dablander, from `http://www.sfs.uni-tuebingen.de/~mfranke/bda+cm2015/homework/sol2.html`. However, it was heavily adjusted, partially based on ideas from Darren Wilkinson, see `https://darrenjw.wordpress.com/2010/08/15/metropolis-hastings-mcmc-algorithms/`, but otherwise based on my own ideas, such as not re-calculating the posterior probability of the current state in each iteration.

Simple Normal Random-Walk MH Also commonly used is the the Simple Normal Random-Walk MH (Liu 2001, p. 114, Jackman 2009, p. 204, Robert & Castella 2010, p. 169). It is quite similar to the previously discussed approach, except for using a normal distribution for $g(\cdot)$, which is illustrated in Equation 10.

$$\begin{aligned} X^* &= X^{(t)} + \epsilon_t, \\ \epsilon_t &\sim \mathcal{N}(0, \Omega) \end{aligned} \tag{10}$$

with Ω being the standard deviation of the normal distribution with $\mu = 0$. As for γ it also holds for Ω, that it needs to be carefully selected in order to tune the algorithm for a given target distribution, which can be done by the user in *ShinyHastings* at the *Algorithm* tab.

Metropolis-Adjusted Langevin Algorithm Albeit employable in most cases, simple random-walks are not necessarily the most efficient proposal types, as Robert & Castella (ibid., p. 186) points out: On the one hand, in symmetric cases the algorithm visits too many superfluous regions. On the other hand, they do not handle well multi-modal distributions, as they tend to remain in local maxima or to show a high rejection rate (ibid., p. 185). Robert & Castella (ibid.) suggests the Metropolis-Adjusted Langevin Algorithm (MALA) by Robert & Tweedie (1996) to overcome this issue.

So, how does MALA differ from the other two random-walk MH algorithms implemented in *ShinyHastings*? As the name suggests, the algorithm combines the Langevin dynamics used to describe molecule movement in physics with the MH algorithm. However, in this case it is used to describe the perturbation around the current state. The Langevin dynamics work with gradient, which direct the movement towards areas that have a higher probability in the target distribution (Robert & Castella 2010, p. 186), i.e. a high $P(\theta^*)$. This is shown in equation 11, where additional to the current state $X^{(t)}$ and some perturbation ϵ, the formula also adds a gradient, the probability of the current state as a weight and an additional, but fixed tuning parameter σ.

$$\begin{aligned} X^* = X^{(t)} + \frac{\sigma^2}{2} \nabla log(P(X^{(t)})) + \sigma\epsilon_t, \\ \epsilon_t \sim g(\cdot), \end{aligned} \tag{11}$$

σ needs to be larger than 0, as the algorithm would not explore the sample space at all, and should not exceed 1. As can be seen, $g(\cdot)$ is not fixed for MALA, the algorithm may come in different flavours, depending on the choice of g. For *Shiny-Hastings* it was implemented using $\mathcal{U}(-\gamma, \gamma)$, just as with the Simple Uniform Random-Walk MH algorithm. The acceptance probability for mala is defined as displayed in equation 12 (cf. Robert & Castella 2010, p. 186).

$$r(X^{(t)}, X^*) = \frac{P(X^*)}{P(X^{(t)})} \frac{J[(X^{(t)} - X^*)/\sigma - \sigma \nabla P(X^*)/2]}{J[(X^* - X^{(t)})/\sigma - \sigma \nabla P(X^{(t)})/2]} \tag{12}$$

As can be seen, the multiplicand takes a more elaborate parameter than the simple random walk proposals, which is why it was separately implemented.

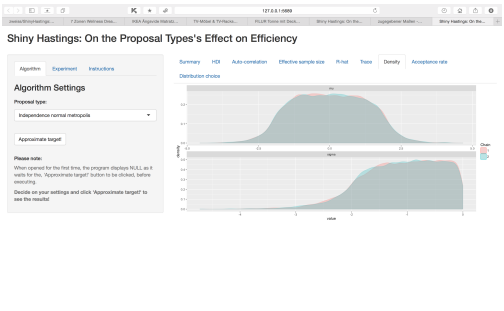
3.3.2 Independence Metropolis Hastings

Independence Normal Metropolis Hastings The independence Metropolis sampler is a special version of the MH algorithm, where the proposed state is not dependent on the current state, as illustrated in equation 13.

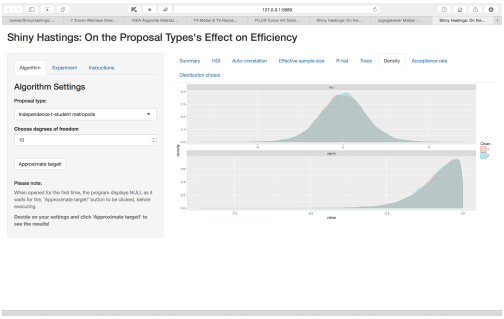
$$X^* \sim g(\cdot) \tag{13}$$

The resulting chain is still a dependent Markov chain due to the acceptance probability (Jackman 2009, p. 204). Robert & Castella (2010) compares this kind of MH algorithm with Accept-Reject sampling, because the proposal is not dependent on the current state. However, as the method still accepts the proposed state probabilistically and in the case of rejection remains at the current state, it still considerably differs from regular Accept-Reject sampling (ibid., p. 175).

How is $g(\cdot)$ chosen? When approximating a posterior distribution, likelihood or prior distributions are popular choices, depending on which resembles the posterior more (Jackman 2009, p. 212). However, normal distributions are also a good choice in many cases. For better comparability with the simple random walk MH algorithm, the latter was chosen for comparability.



(a) Independence Normal Metropolis Hastings



(b) Independence Student Metropolis Hastings

Figure 5: Autocorrelation for sampling from the log normal distribution.

Independence Student Metropolis Hastings Liu (2001, 115f) points out, that the proposal distributions should to be long-tailed in order to lead to robust results. Therefore, a second sort of independence Metropolis algorithm was implemented, using a student's t distribution as $g(\cdot)$, otherwise being identical to the previous one. In fact, both versions of the algorithm seem to work comparably well. Yet, the one using a student's t distribution does tend to produce densities which have a more conclusive maximum around the actual values, as illustrated in figure 5.

4 Conclusion

The *ShinyHastings* web application presented in this article has been designed to serve as an illustration for the effect of various proposal types on the popular MH algorithm. It allows users to playfully explore performance differences across target distributions using a broad range of popular MH types. What kind of differences can in fact be observed using *ShinyHastings* has been illustrated with various examples in the course of this article. All code is

available at `https://github.com/zweiss/ShinyHastings`.

The application also serves as an example how the Shiny framework allows to run powerful R scripts with an appealing interface and, how to overcome some Shiny default settings in order to manage long runtime. However, it was found that the framework grows more unstable with increasingly complex interface constructions and does not necessarily work with all R packages. This leaves room for improvement, however, it does not impair the value Shiny has for R developers who want to share their new programs with a broad audience.

For future work, the choice of proposal types and distributions might still be enhanced. For the latter, it might be desirable to include distributions with different parameters than mean and variance, such as the Poison distribution or the Student's t-distribution. The latter would be especially interesting for further tests with the Metropolized Independence Sampler, which requires a long tailed sampling distribution. Also, a bimodal distribution could be added, to show the full beauty of MALA. This could not be done in time for this version of the program for time reasons, but might be added in future versions.

References

Chib, Siddhartha & Edward Greenberg (1995). "Understanding the Metropolis-Hastings Algorithm". In: *The American Statistician* 49.4, pp. 327–335.

Gelman, Andrew & Donald B. Rubin (1992). "Inference from Iterative Simulation Using Multiple Sequences". In: *Statistical Science* 7.4, pp. 457–472.

Hastings, W. K. (1970). "Monte Carlo Sampling Methods Using Markov Chains and Their Applications". In: *Biometrika* 57.1, pp. 97–109.

Jackman, Simon (2009). *Bayesian Analysis for the Social Sciences*. Wiley Series in Probability and Statistics.

Jurafsky, Daniel & James H. Martin (1999). "Speech and Language Processing. An Introduction to Natural Language Processing, Computational Linguistics and Speech Recognition".

Kruschke, John K. (2015). *Doing Bayesian Data Analysis. A Tutorial with R, JAGS, and Stan*. 2nd ed. Amsterdam: Elsevier.

Liu, Jun S. (2001). *Monte Carlos Strategies in Scientific Computing*. 1st ed. Springer Series in Statistics. New York: Springer.

Martin, Andrew D., Kevin M. Quinn & Jong Hee Park (2011). "MCMCpack: Markov Chain Monte Carlo in R". In: *Journal of Statistical Software* 42.9, pp. 1–21.

Metropolis, Nicholas et al. (1953). "Equation of State Calculations by Fast Computing Machines". In: *Journal of Chemical Physics* 21.6, pp. 1087–1092.

Robert, Christian P. (2016). "The Metropolis-Hastings algorithm". http://arxiv.org/abs/1504.01896v3.

Robert, Christian P. & George Castella (2010). *Introducing Monte Carlo Methods with R*. Use R. Springer.

Robert, Christian P. & R. Tweedie (1996). "Exponential convergence of langevin distributions and their discrete approximations". In: *Bernoulli* 2, pp. 341–363.