Article

Comparing the MCMC Efficiency of JAGS and Stan for the Multi-Level Intercept-Only Model in the Covariance- and Mean-Based and Classic Parametrization

Martin Hecht [1,*], Sebastian Weirich [2] and Steffen Zitzmann [1]

[1] Hector Research Institute of Education Sciences and Psychology, University of Tübingen, 72072 Tübingen, Germany; steffen.zitzmann@uni-tuebingen.de
[2] Institute for Educational Quality Improvement, Humboldt-Universität zu Berlin, 10117 Berlin, Germany; sebastian.weirich@iqb.hu-berlin.de
* Correspondence: martin.hecht@uni-tuebingen.de

Abstract: Bayesian MCMC is a widely used model estimation technique, and software from the BUGS family, such as JAGS, have been popular for over two decades. Recently, Stan entered the market with promises of higher efficiency fueled by advanced and more sophisticated algorithms. With this study, we want to contribute empirical results to the discussion about the sampling efficiency of JAGS and Stan. We conducted three simulation studies in which we varied the number of warmup iterations, the prior informativeness, and sample sizes and employed the multi-level intercept-only model in the covariance- and mean-based and in the classic parametrization. The target outcome was MCMC efficiency measured as effective sample size per second (ESS/s). Based on our specific (and limited) study setup, we found that (1) MCMC efficiency is much higher for the covariance- and mean-based parametrization than for the classic parametrization, (2) Stan clearly outperforms JAGS when the covariance- and mean-based parametrization is used, and that (3) JAGS clearly outperforms Stan when the classic parametrization is used.

Keywords: JAGS; Stan; effective sample size; MCMC efficiency; Bayesian SEM

Citation: Hecht, M.; Weirich, S.; Zitzmann, S. Comparing the MCMC Efficiency of JAGS and Stan for the Multi-Level Intercept-Only Model in the Covariance- and Mean-Based and Classic Parametrization. *Psych* **2021**, *3*, 751–779. https://doi.org/10.3390/psych3040048

Academic Editors: Holmes Finch and Gongjun Xu

Received: 17 October 2021
Accepted: 25 November 2021
Published: 30 November 2021

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Copyright: © 2021 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Bayesian statistics is gaining in popularity in many disciplines and are used for many different purposes, for instance, to include previous knowledge, to estimate otherwise intractable models, to model uncertainty (e.g., [1]), and to stabilize parameter estimates (e.g., [2]).

A popular software platform for Bayesian estimation is the BUGS family including BUGS [3,4] (see [5] for an overview of the history of BUGS), WinBUGS [6], OpenBUGS [7,8], JAGS [9], and NIMBLE [10]. Monnahan et al. (p. 339, [11]) even call BUGS the "workhorse for Bayesian analyses in ecology and other fields for the last 20 years." More recently, the software Stan whose development was inspired by the "pathbreaking programs" BUGS and JAGS (p. 538, [12]) and "motivated by the desire to solve problems that could not be solved in a reasonable time [...] using other packages" (p. 537, [12]) entered the market with promises of higher computational and algorithmic efficiency. Often, the superiority of Stan over JAGS, which is a more modern member of the BUGS family, is claimed to be due to more advanced MCMC algorithms. Whereas JAGS uses conjugate and slice sampling, Stan uses the No-U-Turn Sampler (NUTS; [13]), which is an adaptive variant of Hamiltonian Monte Carlo (HMC; [14]). A comprehensive illustration of NUTS and HMC can be found in the work of Monnahan et al. [11] and more detailed technical descriptions in the works of Nishio and Arakaw [15] and Betancourt [16].

There has been much debate on efficiency differences between Stan and JAGS, and some authors have explored this research question by conducting comparison studies. For instance, Carpenter et al. (p. 10, [17]) found that "Compared to BUGS and JAGS, Stan is

often relatively slow per iteration but relatively fast to generate a target effective sample size." Monnahan et al. (p. 339, [11]) conclude that "[f]or small, simple models there is little practical difference between the two platforms, but Stan outperforms BUGS as model size and complexity grows." which is in line with Gelman et al.'s (p. 538, [12]) statement that "Stan is faster for complex models and scales better than Bugs or Jags for large data sets". However, Grant et al. [18] compared the performance (total time per effective sample size) of various software (including StataStan and JAGS) depending on the number of parameters in the Rasch and hierarchical Rasch model and found that "no one software option dominated" (p. 350, [18]). Additionally, Merkle et al. (p. 2, [19]) report that the original Stan implementation in the package blavaan "was not much faster or more efficient than the JAGS approach.", and Wingfeet [20] concluded that "[n]either JAGS nor Stan came out clearly on top". Further, Bølstad [21] reports mixed results depending on the conjugacy of the priors, with JAGS beating Stan for a fully conjugate hierarchical model. For a completely non-conjugate model with t-distributions instead of normal distribution, the effect reversed, with Stan being much faster.

In summary, the competition between JAGS and Stan has not been finally decided and performance might depend on several factors, for instance, on the model itself, its complexity, number of parameters, priors, and the parametrization.

Purpose and Scope

The purpose of the present work is to contribute to the discussion about the efficiency of JAGS and Stan. To this end, we conducted three simulation studies in which we varied the number of warmup iterations, the informativeness of the prior distributions, the sample sizes, and the model parametrization, and compared the MCMC efficiency operationalized as the effective sample size per second (ESS/s) between JAGS and Stan. The targeted model was the multi-level intercept-only model, which is a popular model in, for example, psychological research and the building block for many more complex multi-level models.

The article is organized into the following sections. First, we describe our Simulation Study 1 including the data generation, the simulation design, the analysis approaches and procedures, and the results of this simulation study. As suggested by anonymous reviewers, we extended the scope of our work by adding Simulation Study 2 (in which we explored a small sample scenario) and Simulation Study 3 (in which we used another model parametrization). Second, we conclude with a discussion of our work. Annotated JAGS/rjags and Stan/rstan code and an example generated data file are provided in the Supplementary Material and Appendices A–G.

2. Simulation Study 1

In this simulation study, the MCMC efficiency of estimating the covariance- and mean-based parametrization of the multi-level intercept-only model with JAGS and Stan is explored.

2.1. Data Generation

The data generating model was the multi-level intercept-only model with overall mean $\mu = 0$, level 2 variance $\sigma^2_\theta = 1$, residual variance $\sigma^2_\epsilon = 1$, $J = 1000$ level 2 units, and $P = 20$ level 1 units:

$$y_{jp} \sim \mathcal{N}(\theta_j, \sigma^2_\epsilon)\,, \tag{1}$$

$$\theta_j \sim \mathcal{N}(\mu, \sigma^2_\theta)\,, \tag{2}$$

where y_{jp} is the pth value of level 2 unit j, and θ_j is unit j's mean parameter. The number of generated data sets (replications) was $N_{\text{repl}} = 1000$. Each of these data sets were analyzed within all 16 design cells of the simulation design.

2.2. Simulation Design

We varied the following factors in our simulation study: software (JAGS, Stan), number of warmup iterations (150, 1000), and prior informativeness (ordered categories A (lower informativeness), B, C, D (higher informativeness)). These factors were fully crossed, yielding 16 design cells. As priors, we used an inverse gamma distribution $IG(\alpha, \beta)$ with shape α and scale β for the variance parameters (σ^2_θ, σ^2_ϵ) and a normal distribution $\mathcal{N}(0, \sigma^2_p)$ with variance σ^2_p for the mean μ. The levels of the prior informativeness factor refer to differing degrees of prior informativeness: A: $\alpha = \beta = 0.001$, σ^2_p = 10,000; B: $\alpha = \beta = 0.01$, $\sigma^2_p = 2500$; C: $\alpha = \beta = 0.1$, $\sigma^2_p = 100$; D: $\alpha = \beta = 1$, $\sigma^2_p = 25$. Thus, the prior informativeness ranges from lower (A) to higher (D).

2.3. Analysis

The simulation study was conducted with the statistical software R [22]. For JAGS [23], the R package rjags [24], and for Stan [25], the R package rstan [26] was used. The analysis model was similar to the data generating model, but we used the covariance- and mean-based implementation of the multi-level intercept-only model [27]:

$$\mathbf{S} \sim \mathcal{W}_P(\mathbf{\Sigma}, J-1)\,, \tag{3}$$

$$\bar{\mathbf{y}} \sim \mathcal{N}_P(\boldsymbol{\mu}, \frac{1}{J}\mathbf{\Sigma})\,, \tag{4}$$

where $\mathbf{S}$ is the sample scatter matrix, $\bar{\mathbf{y}}$ is the sample mean vector, $\mathcal{W}$ is the Wishart distribution, and where $\mathbf{\Sigma}$ and $\boldsymbol{\mu}$ are the model-implied covariance matrix and mean vector:

$$\mathbf{\Sigma} = \begin{pmatrix} \sigma^2_\theta + \sigma^2_\epsilon & & \sigma^2_\theta \\ & \ddots & \\ \sigma^2_\theta & & \sigma^2_\theta + \sigma^2_\epsilon \end{pmatrix}, \tag{5}$$

$$\boldsymbol{\mu} = (\mu \ldots \mu)'\,. \tag{6}$$

The inverse of $\mathbf{\Sigma}$ is:

$$\mathbf{C} = \mathbf{\Sigma}^{-1} = \begin{pmatrix} \frac{(P-1)\sigma^2_\theta + \sigma^2_\epsilon}{\xi} & & -\frac{\sigma^2_\theta}{\xi} \\ & \ddots & \\ -\frac{\sigma^2_\theta}{\xi} & & \frac{(P-1)\sigma^2_\theta + \sigma^2_\epsilon}{\xi} \end{pmatrix}, \tag{7}$$

$$\text{with } \xi = P\sigma^2_\theta\sigma^2_\epsilon + \sigma^4_\epsilon\,. \tag{8}$$

This "covariance- and mean-based approach" [27] is conceptually similar to the "marginal Stan method" [19] as both groups of authors capitalize on the idea of integrating latent variables out of the model likelihood and has, for instance, been shown to be beneficial for MCMC estimation of continuous-time models [28]. Throughout the paper, we consistently use the terms "covariance- and mean-based" and "classic" as defined in the work of Hecht et al. [27], although other labels exist. The former approach is also called "marginal" approach as parameters are integrated out of the likelihood. Formulating models for continuous variables in terms of multivariate normal and Wishart distributions has also previously been described in, for instance, the work of Goldstein [29]. Hecht et al. [27] created the term "classic" to distinguish approaches that include certain parameters from the ones in which they are integrated out.

The parametrization of the Wishart distribution differs in JAGS and Stan. Whereas JAGS's `dwish(`$\mathbf{\Sigma}^{-1}$`, df)` function uses the inverse covariance matrix, Stan's `wishart(df,`$\mathbf{\Sigma}$`)` function uses the covariance matrix. To make the comparison between JAGS and Stan as fair as possible, we avoided time-consuming in-software matrix inversion by passing the parametrization-consistent matrix to the functions, that is, matrix $\mathbf{C}$ from Equation (7) for setting up the model in JAGS and matrix $\mathbf{\Sigma}$ from Equation (5) for setting up the model in

Stan. Avoiding the inverse of large covariance matrices has also been recommended by Goldstein [29].

Each replication ran on one Intel Xeon E5–2687W (3.0 GHz) CPU of a 64-bit Windows Server 2008 system with a total of 48 CPUs, on which we ran 36 replications (each on one core) in parallel. Run times for the following process steps were obtained. For JAGS, the warmup run time [`jags.model()`] and the sampling run time [`jags.samples()`] were determined by the before-after difference of time stamps obtained from the function `Sys.time()`. For Stan, the warmup and sampling run times can be obtained from the console output of the function `sampling()` and are also retrievable from the returned results object (`attributes(fitted@sim$samples[[1]])[["elapsed_time"]]`). Additionally, we recorded the time to set up and compile the Stan model [`stan_model()`, `stanc()`] and also the run time of the function `sampling()` via `Sys.time()` differences. We report all run times in seconds. The number of warmup iterations [set via argument `n.adapt` in `jags.model()` and argument `warmup` in Stan's `sampling()`] was varied in the simulation design (see above), whereas the number of sampling iterations [set via argument `n.iter` in `jags.samples()` and argument `iter` in Stan's `sampling()`] was 100,000. Only these 100,000 sampling iterations served as the basis for computing further statistics (see next paragraph). Hence, the warmup iterations were excluded from further processing and can be considered as omitted "burn-in". Whether omission of additional burn-in was needed or not was determined by visual inspections of trace plots and based on convergence statistics (potential scale reduction (PSR)). One chain per parameter was used without thinning. Starting values were the true parameter values (see Data Generation above).

The effective sample size (ESS) and the PSR were computed with the R package shinystan [30] for both JAGS and Stan samples. The mode of the converged chain served as the parameter estimate. As parameter recovery and precision statistics, bias, root mean squared error (RMSE), and the 95% coverage rate were computed. Our target outcome variable is the *MCMC efficiency*, which we calculated as the ratio of the ESS and the run time (in seconds) of the sampling on one CPU (a similar definition of MCMC efficiency can be found, for example, in the work of Turek et al. [31]). Carpenter et al. (p. 10, [17]) termed this ESS/s (or its inverse) the "most relevant statistic for comparing the efficiency of sampler implementations" because the estimation accuracy is governed by the square root of the ESS, a fact that has also been shown by Zitzmann and Hecht [32] (see also [33]).

2.4. Results

Descriptive statistics (based on the 100,000 sampling iterations) are presented in Table 1. The maximum PSR value is 1.0002 (JAGS) and 1.0003 (Stan), respectively, indicating that all chains had converged. Visual inspection of randomly selected trace plots confirmed trouble-free convergence. Mean ESS values were 74,334 for JAGS and 86,486 for Stan. Hence, Stan produced a higher ESS than JAGS on average.

Average bias is very close to zero ($M_{\text{bias,JAGS}} = -0.0009$, $M_{\text{bias,Stan}} = 0.0005$), RMSEs are practically equal ($M_{\text{RMSE,JAGS}} = 0.0297$, $M_{\text{RMSE,Stan}} = 0.0303$), and the average coverage rates nearly hit 0.95 ($M_{\text{CR,JAGS}} = 0.9608$, $M_{\text{CR,Stan}} = 0.9520$). Bias, RMSE, and coverage rates of JAGS and Stan are very comparable. Thus, both software estimate the parameters of the multi-level intercept-only model similarly well.

Stan needed on average 0.10 s to set up the model [`stan_model()`] and 56.83 s for compilation [`stanc()`]. For the warmup, Stan [console output from rstan's function `sampling()`] took 0.36 s on average, whereas JAGS [rjags's function `jags.model()`] needed 0.87 s on average (however, time to set up the model was included in this warmup run time for JAGS, but not for Stan). For the sampling of the 100,000 values, JAGS [`jags.samples()`] needed 114.99 s and Stan [console output from rstan's function `sampling()`] 52.23 s. Thus, Stan was about twice as fast as JAGS. Considering that a higher ESS was achieved in much shorter time, Stan clearly samples more efficiently than JAGS on average (in our simulation setup). However, in addition to warmup and sampling, Stan's `sampling()` function took

another 215 seconds on average before returning the samples. Thus, users need to wait much longer for the results of the sampling process when using rstan instead of rjags.

Table 1. Descriptive statistics of convergence, precision, recovery, accuracy, and run time by software (Simulation Study 1).

Statistic	Software	*M*	*SD*	Min	Max
Convergence/Precision					
PSR	JAGS	1.0000	0.0000	1.0000	1.0002
	Stan	1.0000	0.0000	1.0000	1.0003
ESS	JAGS	74,334	17,950	46,727	100,000
	Stan	86,486	17,591	27,583	100,000
Recovery/Accuracy					
Bias	JAGS	−0.0009	0.0016	−0.0030	0.0010
	Stan	0.0005	0.0005	−0.0002	0.0010
RMSE	JAGS	0.0297	0.0155	0.0101	0.0474
	Stan	0.0303	0.0164	0.0096	0.0489
Coverage rate 95%	JAGS	0.9608	0.0054	0.9530	0.9670
	Stan	0.9520	0.0075	0.9420	0.9620
Run time (s)					
Warmup	JAGS	0.87	0.81	0.13	21.95
	Stan	0.36	0.24	0.08	2.05
Sampling	JAGS	114.99	7.48	89.55	155.42
	Stan	52.23	8.23	30.18	97.65
Translation	Stan	0.10	0.02	0.06	0.37
Compilation	Stan	56.83	7.29	42.09	132.18
`sampling()`	Stan	214.65	12.86	185.62	270.97

Note. Run time for rstan's `sampling()` function is the additional time that this function runs besides warmup and sampling.

Figure 1 shows the MCMC efficiency (i.e., the effective sample size produced in one second) split by the levels of the simulation factors and the model parameters. Additional to these marginal mean MCMC efficiencies, we present mean MCMC efficiencies for the simulation factors and model parameters split by software JAGS and Stan in Figure 2 to investigate interaction effects. The overall mean MCMC efficiency was 1180 ESS/s. Considering software, the average MCMC efficiency is 649 for JAGS and 1712 for Stan. Thus, Stan outperforms JAGS by roughly a factor of two and a half on average. Mean MCMC efficiency is lower for 150 warmup iterations ($M_{\text{warmup}=150} = 1034$) than for 1000 warmup iterations ($M_{\text{warmup}=1000} = 1327$). From Figure 2 it becomes clear that there is an interaction between software and number of warmup iterations. Whereas JAGS's MCMC efficiency is approximately equal for 150 and 1000 warmup iterations ($M_{\text{warmup}=150,\text{JAGS}} = 652$, $M_{\text{warmup}=1000,\text{JAGS}} = 647$), Stan profits very much from more warmup iterations ($M_{\text{warmup}=150,\text{Stan}} = 1416$, $M_{\text{warmup}=1000,\text{Stan}} = 2007$). The prior informativeness has practically no effect on the MCMC efficiency ($M_{\text{A}} = 1180$, $M_{\text{B}} = 1175$, $M_{\text{C}} = 1181$, $M_{\text{D}} = 1186$). With respect to the three model parameters, the MCMC efficiency differs on average ($M_{\mu} = 1243$, $M_{\sigma^2_\theta} = 1053$, $M_{\sigma^2_\epsilon} = 1246$), and an interaction with software is evident. Whereas JAGS's MCMC efficiency for both variance parameters is equal ($M_{\sigma^2_\theta,\text{JAGS}} = 539$, $M_{\sigma^2_\epsilon,\text{JAGS}} = 540$), it is higher for the mean ($M_{\mu,\text{JAGS}} = 869$). For Stan, the picture is different. Here, μ and σ^2_θ show less MCMC efficiency ($M_{\mu,\text{Stan}} = 1616$, $M_{\sigma^2_\theta,\text{Stan}} = 1567$) than the residual variance σ^2_ϵ ($M_{\sigma^2_\epsilon,\text{Stan}} = 1952$).

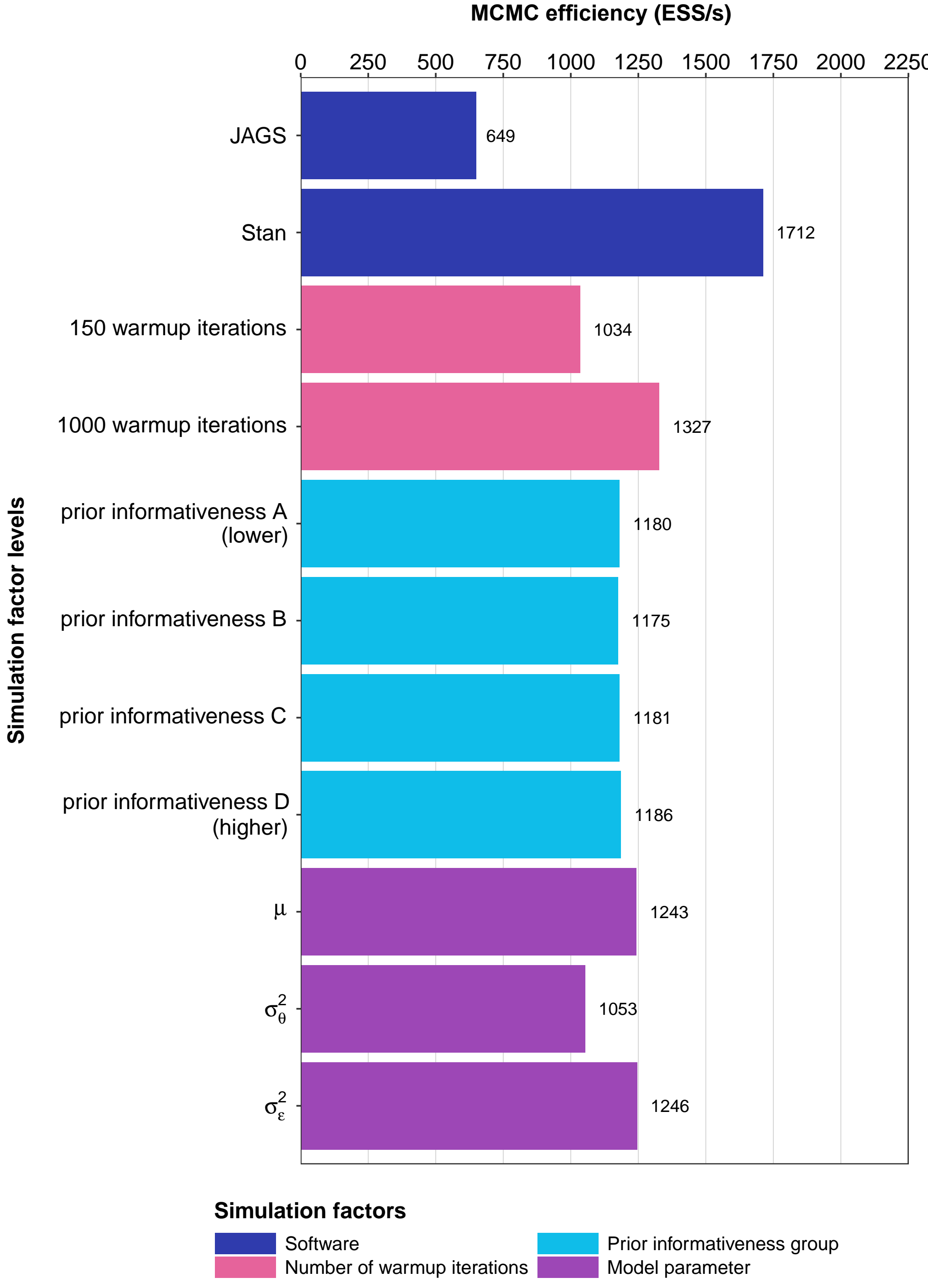

Figure 1. Simulation Study 1: ESS performance (ESS/s) of simulation factors software (JAGS/Stan), number of warmup iterations (150/1000), prior informativeness (A/B/C/D), and model parameters (μ, σ^2_θ, σ^2_ε). Number of level 2 units: $J = 1000$. Number of level 1 units: $P = 20$. Covariance- and mean-based model parametrization.

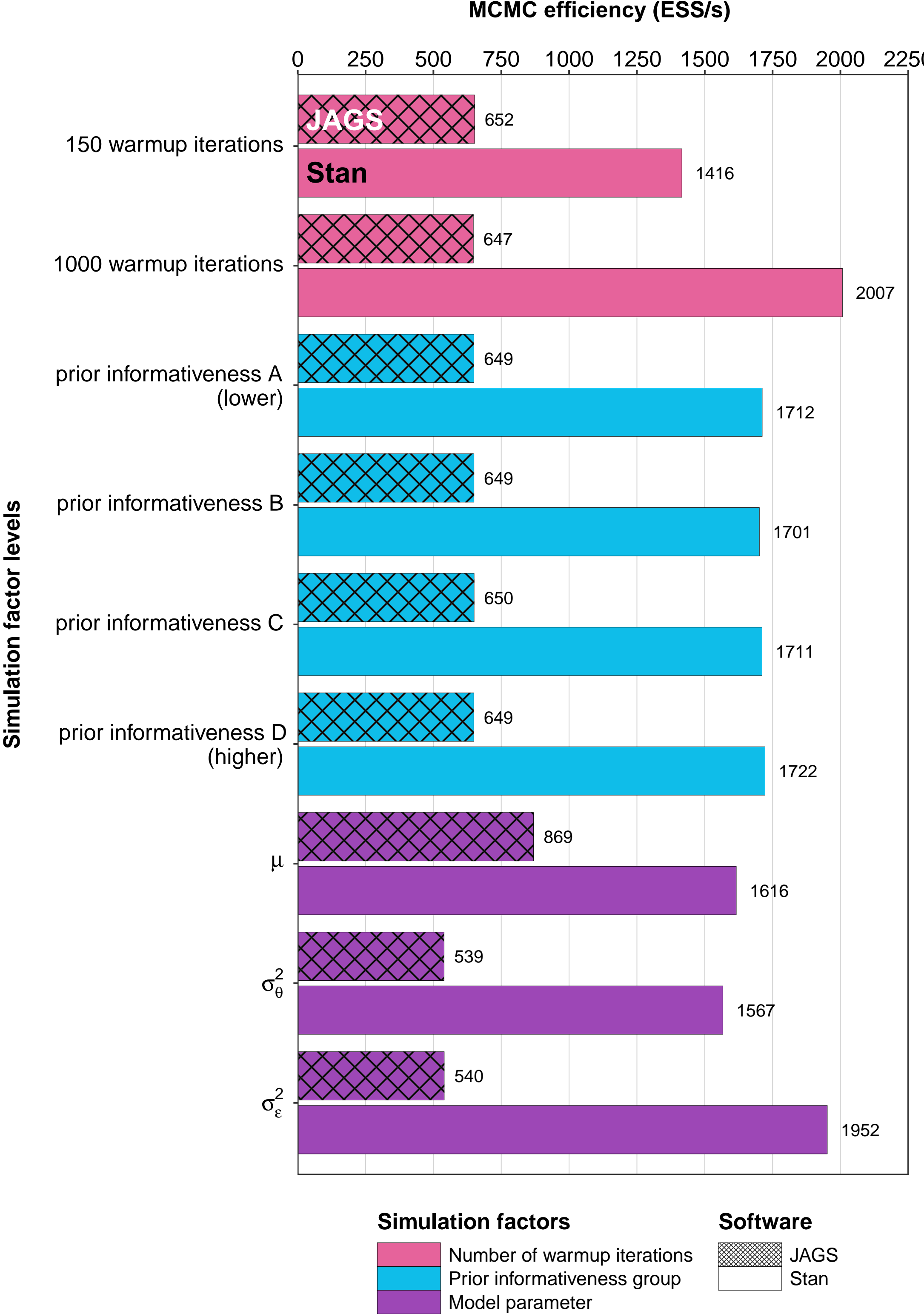

Figure 2. Simulation Study 1: ESS performance (ESS/s) of number of warmup iterations (150/1000), prior informativeness (A/B/C/D), and model parameters (μ, σ^2_θ, σ^2_ε) by software (JAGS/Stan). Number of level 2 units: $J = 1000$. Number of level 1 units: $P = 20$. Covariance- and mean-based model parametrization.

Effect sizes η^2 for the simulation factors are presented in Table 2. Software exhibits the by far highest variance explanation (68.4%). Warmup iterations and parameter explain 5.2% and 2.0% of the variance in MCMC efficiency, respectively, whereas variance explanation

by prior informativeness is essentially zero. Interactions with above zero variance explanation are Software × Warmup Iterations (5.4%), Software × Parameter (4.5%), Warmup Iterations × Parameter (1.0%), and Software × Warmup Iterations × Parameter (1.0%).

Table 2. Effect size η^2 for the simulation factors (Simulation Study 1).

Factor	η^2
Software	0.684
Warmup Iterations	0.052
Prior Informativeness	0.000
Parameter	0.020
Software × Warmup Iterations	0.054
Software × Prior Informativeness	0.000
Software × Parameter	0.045
Warmup Iterations × Prior Informativeness	0.000
Warmup Iterations × Parameter	0.010
Prior Informativeness × Parameter	0.000
Software × Warmup Iterations × Prior Informativeness	0.000
Software × Warmup Iterations × Parameter	0.010
Software × Prior Informativeness × Parameter	0.000
Warmup Iterations × Prior Informativeness × Parameter	0.000
Software × Warmup Iterations × Prior Informativeness × Parameter	0.000

Note. Dependent variable: MCMC efficiency (ESS/s).

In summary, both software estimate the multi-level intercept-only model equally well, but Stan outperforms JAGS in the production of effective sample size per time unit. Further, Stan profits from more warmup iterations, the prior informativeness is practically not related to the MCMC efficiency, and the MCMC efficiency differs between model parameters.

3. Simulation Study 2: Small Sample Size

In this simulation study, the MCMC efficiency of estimating the covariance- and mean-based parametrization of the multi-level intercept-only model with JAGS and Stan is explored for a small sample size scenario. The simulation design and the analysis strategy were similar to Simulation Study 1. The data generation was similar as well, except that the number of level 2 units was reduced to $J = 100$ and the number of level 1 units to $P = 5$.

The overall mean efficiency in this small sample scenario is 8343 ESS/s and thus roughly seven times higher than in Simulation Study 1 where sample sizes were larger ($J = 1000$, $P = 20$). Investigating the mean MCMC efficiencies split by the simulation factors, model parameters, and software (Figure 3) yields basically the same pattern as in Simulation Study 1. Stan outperforms JAGS; however, JAGS' underperformance is not as pronounced as in the large sample scenario. Concerning warmup, the figure again shows that JAGS does not profit from more warmup iterations, whereas Stan does. Prior informativeness exhibits no clear effect. An interaction of parameter and software can again be identified. Whereas JAGS performs better in efficiently estimating μ than σ^2_θ and σ^2_ϵ, Stan is approximately equally efficient in estimating all three model parameters.

In summary, reducing the sample size resulted in higher overall MCMC efficiency, but software differences in MCMC efficiency remained, although JAGS caught up somewhat to Stan.

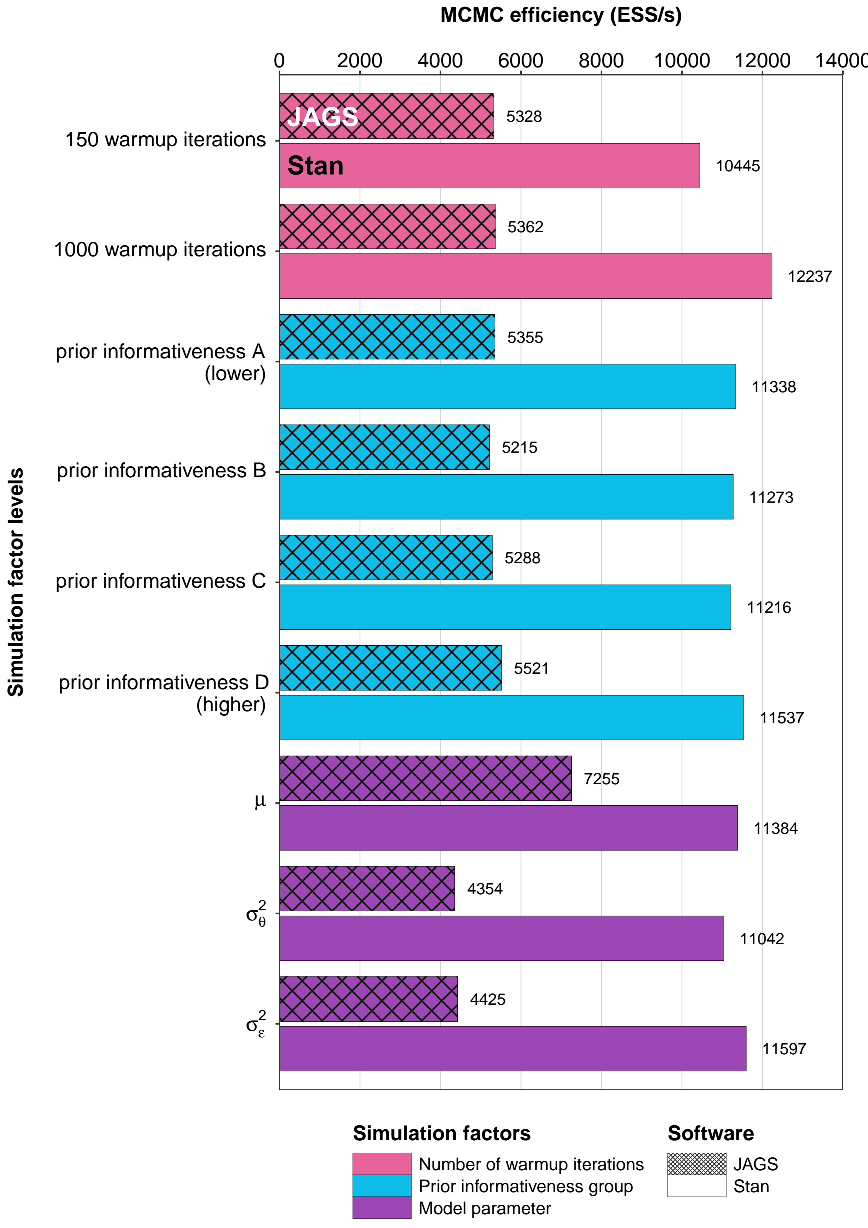

Figure 3. Simulation Study 2 (small sample size): ESS performance (ESS/s) of number of warmup iterations (150/1000), prior informativeness (A/B/C/D), and model parameters (μ, σ^2_θ, σ^2_ϵ) by software (JAGS/Stan). Number of level 2 units: $J = 100$. Number of level 1 units: $P = 5$. Covariance- and mean-based model parametrization.

4. Simulation Study 3: Classic Parametrization

In this simulation study, the MCMC efficiency of estimating the classic parametrization of the multi-level intercept-only model with JAGS and Stan is explored. The data generation and the simulation design were similar to Simulation Study 1. For the analysis model, the classic parametrization was now chosen, in which the parameters of the level 2 units (θ_j in Equations (1) and (2)) were part of the model formulation and thus needed to be sampled (see [27] for further details on the differences between the classic and the covariance- and mean-based parametrization of the multi-level intercept-only model). With $J = 1000$ level 2 units, this meant that 1000 additional parameters needed to be sampled (an increase by a factor of 334 compared to Simulation Study 1). Hence, to keep the simulation within manageable boundaries, the number of sampling iterations was reduced to 10,000.

Compared to Simulation Studies 1 and 2 (with the covariance- and mean-based parametrization), two major differences emerged: (1) The overall mean efficiency (137 ESS/s) is by far lower, and (2) the software rank order reversed: When employing the classic parametrization, JAGS is much more efficient than Stan (see Figure 4). Again, Stan profits from more warmup iterations, whereas JAGS does not, and prior informativeness has no effect on MCMC efficiency. Concerning the parameters, there is no clear differential picture (i.e., within software, the parameters are estimated with roughly the same efficiency), although Stan seems to have problems with estimating the residual variance σ^2_ϵ (with just 6 ESS/s).

In summary, choosing the classic instead of the covariance- and mean-based implementation resulted in lower overall MCMC efficiency, and JAGS clearly outperformed Stan.

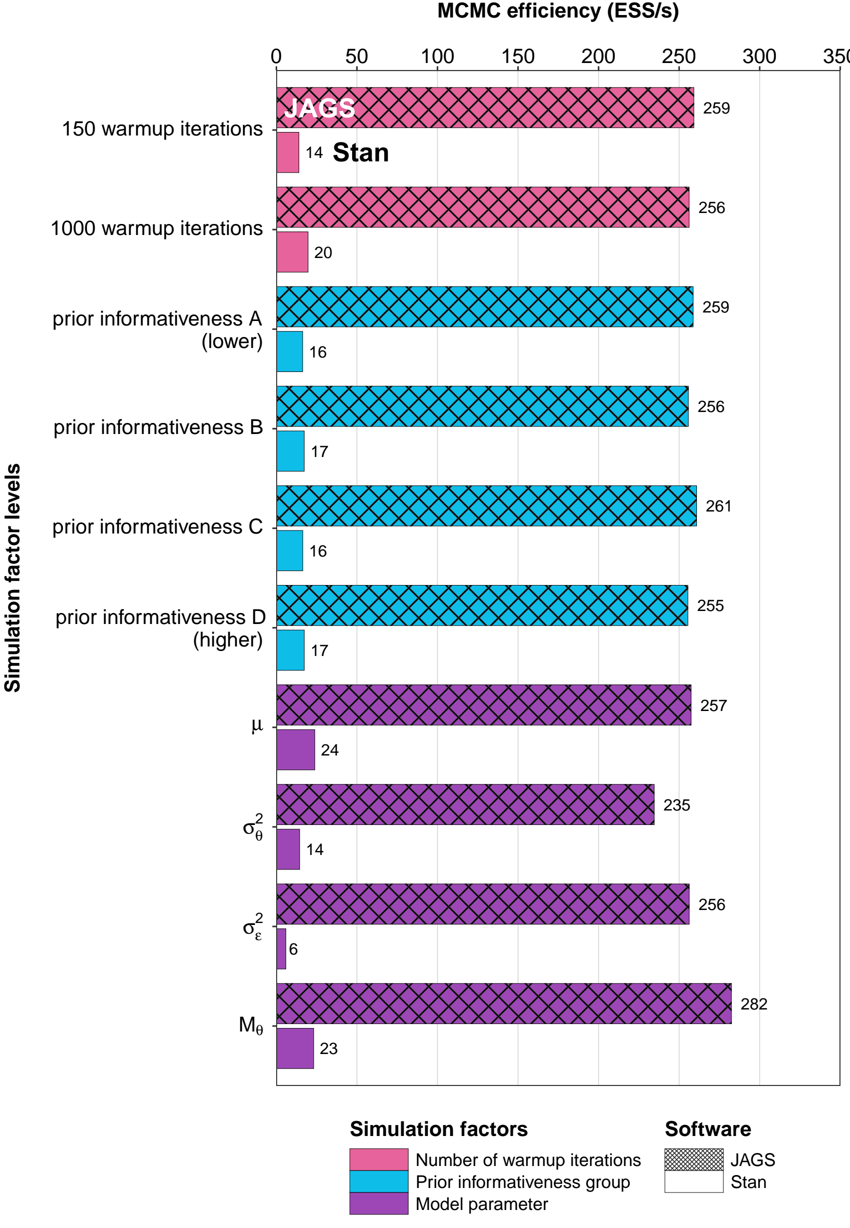

Figure 4. Simulation Study 3 (classic parametrization): ESS performance (ESS/s) of number of warmup iterations (150/1000), prior informativeness (A/B/C/D), and model parameters (μ, σ^2_θ, σ^2_ϵ) by software (JAGS/Stan). M_θ is the average MCMC efficiency of level 2 units parameters θ_j. Number of level 2 units: $J = 1000$. Number of level 1 units: $P = 20$. Classic model parametrization.

5. Discussion

With this study, we want to contribute empirical results to the discussion about the sampling efficiency of JAGS and Stan. In our limited simulations, Stan outperformed (i.e., exhibited a higher ESS/s) JAGS for the covariance- and mean-based parametrization of the multi-level intercept-only model. However, when the classic parametrization was chosen, JAGS outperformed Stan. Additionally, we found that Stan profited from more warmup iterations and that prior informativeness had no effect on the MCMC efficiency.

The results from our simulation study most certainly will not generalize to other models (or model parametrizations), conditions, or other values/levels of our simulation factors.

Our model was a very simple one. Stan is often said to gain in efficiency with increasing model complexity (e.g., [11,12]). We found higher efficiency (roughly by a factor of 2.5) already for a simple model. It would be interesting whether Stan outperforms JAGS even more for more complex multi-level models and other relevant models for psychological research.

Besides the model itself, the parametrization of the model is crucial as well. We showed that with the classic parametrization of the multi-level intercept-only model, JAGS clearly outperformed Stan. As the classic parametrization is presumably the most intuitive and easiest to implement for psychologists, the software recommendation clearly leans towards JAGS here. However, as shown, users can speed up their analyses (i.e., more efficiently estimate the model parameters) by switching to the covariance- and mean-based parametrization (a comprehensive tutorial on how to set up this parametrization is given by Hecht et al. [27]). For this parametrization, Stan clearly outperforms JAGS and should be the software of choice (if MCMC efficiency is the target criterion). Our results are in line with Merkle et al.'s [19] results who reported superiority of their "new Stan approach" which is conceptually similar to Hecht et al.'s [27] covariance- and mean-based approach. The efficiency of this new approach was even so convincing that the authors of R package blavaan made it their new default method (p. 12, [19]). However, in contrast to Hecht et al. ([27], Equations (13) and (14)), Merkle et al. ([19], Equations (5) and (6)) use multivariate normal marginal distributions of the sample value vectors (presumably because it might be a more flexible and convenient approach, for instance for handling missing data). Future research could investigate performance differences between the Wishart and the multivariate normal parametrization. From our experience with JAGS (not published), multivariate normal modeling of the sample value vectors was much slower than the Wishart modeling of the sample scatter matrix, but this might be different for Stan. Future research could compare more parametrizations of multi-level models.

An anonymous reviewer pointed us to an interesting advantage of marginalized model parametrizations which is detailed in the work of Nielsen et al. [34]. In the classic parametrization that includes random effects, positive within-cluster correlations are assumed. The covariance-based parametrization, in which random effects are integrated out, is more general because negative correlations are allowed as well.

Another approach to improve the efficiency of MCMC sampling is to formulate the model in a way that autocorrelation in the chains is reduced. Various strategies for various models have been proposed (e.g., [35–37]). According to Monnahan et al. (p. 344, [11]), "MCMC efficiency for hierarchical models depends on the random effects parameterization, with the *centered* and *non-centered* complementary forms being useful for a broad class of models". In our simulations, we solely used the centered form (Equation (2)) because it is presumably the "natural" form researchers obtain when they translate their model equations into code. In the alternative non-centered form, the random effects (θ_j) would not be modeled directly, instead as $\theta_j = \mu + \sigma Z$ with $Z \sim \mathcal{N}(0, 1)$. Results from Monnahan et al.'s (p. 346, [11]) comparisons of JAGS and Stan suggest that Stan is "more sensitive to the parameterization of the random effects, suggesting analysts use non-centered parameterizations to improve performance". Future research could generate

further evidence on how to improve MCMC efficiency by model reformulations and explore which software profits most.

Concerning sample sizes, other studies (e.g., [11]) report that Stan's relative efficiency over JAGS increases with increasing sample size. Our results are in line with this finding (for the covariance- and mean-based parametrization); for our small sample size scenario, Stan did not outperform JAGS as much as in the large sample size scenario. As we only had two sample size scenarios (and varied sample size only for the covariance- and mean-based parametrization), generalizations are limited. Future research could investigate the dependency of sample sizes on the efficiency and the moderating effect of sample size on the difference in efficiency between both software in more detail, especially also with respect to the model parametrization. Additionally, the relative performance of different methods might depend on the ratio of the variance components in the simulation model.

We used a high number of iterations to achieve sensibly sized absolute runtimes and to obtain reliable ESS estimates [38]. We assume that the MCMC efficiency (ESS/s) is constant over the course of sampling. Hence, given the chain has converged, the length of the sampling should have no effect on the MCMC efficiency.

We limited our study to two popular Bayesian software in psychological research, namely JAGS and Stan. Other Bayesain software packages, for instance, NIMBLE [10], PyMC3 [39], and LaplacesDemon [40], exist and have been the focus of research (e.g., [38,41–43]). Future software comparisons should take these packages even more strongly into account.

The effect of the warmup iterations on Stan's MCMC efficiency is in line with the functioning of the NUTS algorithm as this algorithm needs to tune the step size to achieve a target acceptance rate and to tune the mass matrix whose function is to transform the posterior to have a simpler geometry for sampling (e.g., [11]). As the optimization of the step size and the mass matrix are mutually dependent, a sufficiently long warmup is needed [11]. We had only two warmup iteration sizes (150 and 1000), with 150 being the lowest default size for one warmup cycle (p. 150, [44]). An anonymous reviewer pointed out that the marginal variances of the parameters in our model are rather small; therefore, it is not surprising that one warmup cycle was not sufficient for optimization. In future research, it would be interesting to explore the shape of the functional dependency of the MCMC efficiency on the number of warmup iterations and determine areas of diminishing marginal utility to derive rule-of-thumb thresholds for sufficient warmup of the NUTS algorithm. Additionally, one could facilitate warmup optimization by programming all parameters "so that they have unit scale and so that posterior correlation is reduced; [...] For Hamiltonian Monte Carlo, this implies a unit mass matrix, which requires no adaptation as it is where the algorithm initializes." (p. 266, [45]). Further, the shown warmup dependency should be taken into consideration when comparing results from studies that differed in this aspect. Merkle et al. (p. 8, [19])—who used 300 warmup iterations for Stan—already pointed to this problem and concluded that therefore "the ESS/s metric is somewhat crude".

We found no effect of the prior informativeness on the MCMC efficiency. However, this result is just valid for the specific four variations of prior informativeness (and all other specifications) in our simulation setup and might not generalize. In fact, in simulation runs with much lower informativeness (not reported), Stan's MCMC efficiency was lower. Future research could investigate the prior informativeness effect on efficiency with a wider range of informativeness. Additionally, in our simulation, the amount of data was rather high, marginalizing the effect of the priors. In scenarios with less data, prior effects might arise.

The conjugacy of priors may play a role in MCMC efficiency as well. Using conjugate priors is usually considered computationally more efficient than using non-conjugate priors. Some software/algorithms might even profit from conjugate priors more than others. For instance, results from Bølstad [21] suggest that JAGS performs better than Stan when priors are conjugate and worse when priors are non-conjugate. In our simulations, we used conjugate priors for all parameters in all conditions, leaving prior conjugacy a constant.

Thus, we cannot contribute to the discussion of the effect of prior conjugacy on MCMC efficiency. Researchers could pick up this interesting topic in the future.

Although Stan was clearly more efficient than JAGS in the sampling phase, the total time users encounter until samples are returned depends on additional steps. Stan models need to be compiled into C++ code prior to sampling, which was considerable in our study (on average, model compilation took longer than the sampling of 100,000 iterations). Of course, once compiled, models may be reused to avoid the extensive compilation time. Still, users who run a model for the first time (or are not aware that previously compiled models may be reused) must afford the compilation time, and rstan's primary user-level function stan() that includes all processes may mask the opportunity for reusing compiled models and contribute to a lengthy user experience. Further, we encountered a relatively huge consumption of additional time besides warmup and sampling by rstan's `sampling()` function. In fact, the additional time was about twice the time for all other steps combined. We are not sure what this function needs this additional time for. According to an anonymous reviewer, the additional time is partly due to calculating ESS and PSR. As this needs to be done for JAGS as well, fair software comparisons would also need to take this into account. Maybe future versions of rstan's `sampling()` function may include an option to return the samples directly after sampling or explicitly report the time needed for additional calculations of convergence and precision statistics.

To conclude, in our specific study setup, the picture concerning MCMC efficiency was mixed. Stan clearly outperformed JAGS when the covariance- and mean-based parametrization of the multi-level intercept-only model was used and JAGS clearly outperformed Stan when the classic parametrization was used. In both software, MCMC efficiency is much higher for the covariance- and mean-based parametrization than for the classic parametrization.

Supplementary Materials: The following are available online at https://www.mdpi.com/article/10.3390/psych3040048/s1, The supplementary materials include an exemplary generated data set in Rdata format, JAGS and Stan code for the multi-level intercept-only model in the covariance- and mean-based and in the classic parametrization, R code to run the models with rjags and rstan, and R code to run simulations.

Author Contributions: The authors declare the following contributions (as defined by http://credit.niso.org (accessed on 30 November 2021)) to this article: M.H.: conceptualization, data curation, formal analysis, investigation, methodology, project administration, software, supervision, validation, visualization, writing: original draft, writing—review and editing; S.W.: conceptualization, data curation, formal analysis, investigation, methodology, software, validation, writing—review and editing; S.Z.: conceptualization, methodology, supervision, validation, writing—review and editing. All authors have read and agreed to the published version of the article.

Funding: The authors received no financial support for the research, authorship, and/or publication of this article.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: In this study, only generated data was used. The data generating code is provided in Appendix G.

Conflicts of Interest: The authors declare no conflict of interest.

Appendix A. rjags Code

```
############################################################################
# R code for running the multi-level intercept-only model with JAGS/rjags #
############################################################################

## used R version: 4.0.5 (R Core Team, 2021)
# => get latest version: https://cran.r-project.org/

## install JAGS from
# https://sourceforge.net/projects/mcmc-jags/files
# used JAGS version: 4.3.0 (Plummer, 2017)

## load/install packages
# used rjags version: 4-10 (Plummer, 2019)
# => get latest rjags version: install.packages( "rjags" )
library( rjags )
# used shinystan version: 2.5.0 (Gabry, 2018)
# => get latest shinystan version: install.packages( "shinystan" )
library( shinystan )

## bugs files
# covariance-/mean-based parametrization
bugs.file <- c("https://figshare.com/ndownloader/files/31595090")
# for classic parametrization use:
# bugs.file <- c("https://figshare.com/ndownloader/files/31595099")

## open bugs file
bugs.file <- url( bugs.file )

## load simulated example data
(load(url("https://figshare.com/ndownloader/files/31595084")))
# D: data matrix in wide format:
#    rows:    level 1 units (e.g., measurement occasions)
#    columns: level 2 units (e.g., persons)
#    cells:   values
# J: number of level 2 units
# P: number of level 1 units


#####################
### preprocessing ###
#####################

## compute sample mean vector and sample scatter matrix from data
# sample mean vector y.bar
y.bar <- matrix( rowMeans( D ), P, 1 )
# column vector of ones
ones.vec <- matrix( 1, J, 1 )
# sample scatter matrix
S <- ( D - y.bar %*% t(ones.vec) ) %*% t( ( D - y.bar %*% t(ones.vec) ) )
```

```
## data list for JAGS
data.list <- list()
data.list <- c( data.list, list( "S"=S ) )
data.list <- c( data.list, list( "y.bar"=y.bar ) )
data.list <- c( data.list, list( "J"=J ) )
data.list <- c( data.list, list( "P"=P ) )
data.list <- c( data.list, list( "D"=D ) ) # only needed for classic parametrization

## function to generate random starting values
# (starting values are true values from data generation: mu = 0, var.theta = 1, var.eps = 1)
generate.startval <- function(){
   startval <- list()
   startval <- c( startval, list( "mu"     = 0   ) )
   startval <- c( startval, list( "prec.theta" = 1/1 ) )
   startval <- c( startval, list( "prec.eps" = 1/1 ) )
   return( startval )
}
# optional for classic parametrization:
# add starting values for parameters of level 2 units (theta)

# set starting values for the parameters (one chain each)
inits <- sapply( 1:1, function (chain.nr,l) l, generate.startval(), simplify=FALSE )

# set seeds for the chain
inits[[1]]$".RNG.name" <- "base::Wichmann-Hill"
inits[[1]]$".RNG.state" <- c( 11076, 3733, 19174 )



##############
### warmup ###
##############

# warmup iterations
warmup.iterations <- 1000

# warmup start time
start.time.warmup <- Sys.time()

# set up & warm up JAGS model
ini <- jags.model( file=bugs.file, data=data.list,
                 n.chains=1, n.adapt=warmup.iterations,
                 inits=inits, quiet=TRUE )

# warmup runtime
( runtime.warmup <- Sys.time() - start.time.warmup )

# close bugs file
close( bugs.file )
```

```
################
### sampling ###
################

# sampling iterations
sampling.iterations <- 100000

# sampling start time
start.time.sampling <- Sys.time()

# sampling
res <- jags.samples( ini, variable.names=c("mu","sigma2.theta","sigma2.eps"),
                   n.iter=sampling.iterations, thin=1, type='trace', progress.bar = NULL )
# optional for classic parametrization:
# add "theta" to argument variable.names

# sampling runtime
( runtime.sampling <- Sys.time() - start.time.sampling )

# set time units of sampling runtime to seconds (needed below to compute ESS/s)
units( runtime.sampling ) <- "secs"


######################
### postprocessing ###
######################

# create shinystan object
samples <- cbind( matrix( res$mu[1,,1], ncol=1 ),
                 matrix( res$sigma2.theta[1,,1], ncol=1 ),
                 matrix( res$sigma2.eps[1,,1], ncol=1 ) )
colnames( samples ) <- c( "mu", "sigma2.theta", "sigma2.eps" )
shinystan.obj <- as.shinystan( list( samples ) )

# retrieve effective sample size (ESS)
( ESS <- retrieve( shinystan.obj, "ess" ) )

# calculate MCMC efficiency (ESS/s)
( MCMC.efficiency <- ESS / as.numeric( runtime.sampling ) )
```

Appendix B. JAGS Code (Covariance- and Mean-Based Parametrization)

```
# JAGS model code for the multi-level intercept-only model
# in the covariance- and mean-based parametrization

model {

   # distributional assumption of sample scatter matrix (Equation 3)
   S ~dwish( C , J-1 )

   ## construction of precision matrix C (Equation 7)
   # main diagonal
   for ( p in 1:P ){
      C[p,p] <-((P-1)*sigma2.theta+sigma2.eps)/ksi
   }
   # lower/upper triangle
   for (k in 2:P) {
      for (l in 1:(k-1)) {
         C[k,l] <--sigma2.theta/ksi
         C[l,k] <-C[k,l]
      }
   }
   # ksi (Equation 8)
   ksi <-P * sigma2.theta * sigma2.eps + sigma2.eps^2

   # precision matrix of means
   Cm <-J * C

   # distributional assumption of sample means (Equation 4)
   y.bar ~dmnorm( mu.vec, Cm )

   # construction of mu vector
   for ( p in 1:P ){
      mu.vec[p] <-mu
   }

   # prior distribution for mu
   mu ~dnorm( 0, 1/10000 )

   # prior distribution for 1/sigma2.theta
   prec.theta ~dgamma( 0.001, 0.001 )
   sigma2.theta <-1/prec.theta

   # prior distribution for 1/sigma2.eps
   prec.eps ~dgamma( 0.001, 0.001 )
   sigma2.eps <-1/prec.eps
}
```

Appendix C. JAGS Code (Classic Parametrization)

```
# JAGS model code for the multi-level intercept-only model
# in the classic parametrization

model {

  # loop over level 2 units
  for (j in 1:J) {

    # loop over level 1 units
    for (p in 1:P) {
      # distributional assumption of observations (Equation 1)
      D[p,j] ~dnorm( theta[j], prec.eps )
    }

    # distributional assumption of the person parameters (Equation 2)
    theta[j] ~dnorm( mu, prec.theta )
  }

  # prior distribution for mu
  mu ~dnorm( 0, 1/10000 )

  # prior distribution for 1/sigma2.theta
  prec.theta ~dgamma( 0.001, 0.001 )
  sigma2.theta <-1/prec.theta

  # prior distribution for 1/sigma2.eps
  prec.eps ~dgamma( 0.001, 0.001 )
  sigma2.eps <-1/prec.eps

}
```

Appendix D. rstan Code

```
##############################################################################
# R code for running the multi-level intercept-only model with Stan/rstan #
##############################################################################

## used R version: 4.0.5 (R Core Team, 2021)
# => get latest version: https://cran.r-project.org/

## load/install packages
# used rstan version: 2.21.2 (Stan Development Team, 2020)
# => get latest rstan version: install.packages( "rstan" )
library( rstan )
# used shinystan version: 2.5.0 (Gabry, 2018)
# => get latest shinystan version: install.packages( "shinystan" )
library( shinystan )

## working directory
# (!!! needs to include file 02b_multilevel_intonly_model_covmeanbased.stan !!!)
wd <- "SET_YOUR_WORKING_DIRECTORY_HERE___needs_to_include_file___02b_multilevel_intonly_model_covmeanbased.stan"
# for classic parametrization include: 02c_multilevel_intonly_model_classic.stan

## load simulated example data
(load(url("https://figshare.com/ndownloader/files/31595084")))
# D: data matrix in wide format:
#    rows:    level 1 units (e.g., measurement occasions)
#    columns: level 2 units (e.g., persons)
#    cells:   values
# J: number of level 2 units
# P: number of level 1 units


#####################
### preprocessing ###
#####################

## compute sample mean vector and sample scatter matrix from data
# sample mean vector y.bar
y.bar <- matrix( rowMeans( D ), P, 1 )
# column vector of ones
ones.vec <- matrix( 1, J, 1 )
# sample scatter matrix
S <- ( D - y.bar %*% t(ones.vec) ) %*% t( ( D - y.bar %*% t(ones.vec) ) )
```

```
## data list for Stan
data.list <- list()
data.list <- c( data.list, list( "S"=S ) )
data.list <- c( data.list, list( "ybar"=as.vector( y.bar ) ) )
data.list <- c( data.list, list( "J"=J ) )
data.list <- c( data.list, list( "P"=P ) )
data.list <- c( data.list, list( "D"=D ) ) # only needed for classic parametrization


#########################
### model translation ###
#########################

# translation start time
start.time.translation <- Sys.time()

# translate model
translated <- stanc( file = file.path( wd, "02b_multilevel_intonly_model_covmeanbased.stan" ), isystem = wd )
# for classic parametrization use: 02c_multilevel_intonly_model_classic.stan
# translated <- stanc( file = file.path( wd, "02c_multilevel_intonly_model_classic.stan" ), isystem = wd )

# translation runtime
( runtime.translation <- Sys.time() - start.time.translation )


#########################
### model compilation ###
#########################

# compilation start time
start.time.compilation <- Sys.time()

# compile model
compiled <- stan_model( stanc_ret = translated, auto_write = TRUE, isystem = wd )

# compilation runtime
( runtime.compilation <- Sys.time() - start.time.compilation )


################################################
### warmup/sampling with function sampling() ###
################################################

# samplingfunction start time
start.time.samplingfunction <- Sys.time()

# warmup/sampling iterations
warmup.iterations <- 1000
sampling.iterations <- 100000
```

```
# warmup/sampling
res <- sampling( object = compiled, data = data.list, chains = 1, cores = 1,
                iter = warmup.iterations+sampling.iterations, warmup = warmup.iterations,
                init = list( list( mu = 0, sigma2theta = 1, sigma2eps = 1 ) ),
                seed = 12345 )
# optional for classic parametrization:
# add starting values (argument init) for parameters of level 2 units (theta)

# samplingfunction runtime
( runtime.samplingfunction <- Sys.time() - start.time.samplingfunction )


######################
### postprocessing ###
######################

# times reported by rstan
runtimes <- attributes( res@sim$samples[[1]] )[["elapsed_time"]]

# time for warmup
( runtime.warmup <- runtimes[["warmup"]] )

# time for sampling
( runtime.sampling <- runtimes[["sample"]] )

# create shinystan object
samples <- cbind( matrix( res@sim[["samples"]][[1]]$mu[(warmup.iterations+1):(warmup.iterations+sampling.iterations)], ncol=1 ),
                 matrix( res@sim[["samples"]][[1]]$sigma2theta[(warmup.iterations+1):(warmup.iterations+sampling.iterations)], ncol=1 ),
                 matrix( res@sim[["samples"]][[1]]$sigma2eps[(warmup.iterations+1):(warmup.iterations+sampling.iterations)], ncol=1 ) )
colnames( samples ) <- c( "mu", "sigma2.theta", "sigma2.eps" )
shinystan.obj <- as.shinystan( list( samples ) )

# retrieve effective sample size (ESS)
( ESS <- retrieve( shinystan.obj, "ess" ) )

# calculate MCMC efficiency (ESS/s)
( MCMC.efficiency <- ESS / as.numeric( runtime.sampling ) )
```

Appendix E. Stan Code (Covariance- and Mean-Based Parametrization)

```
// Stan model code for the multi-level intercept-only model
// in the covariance- and mean-based parametrization

data {
  matrix[20,20] S; // sample scatter matrix
  vector[20] ybar; // sample mean vector
  int<lower=1> J; // number of level 2 units
  int<lower=1> P; // number of level 1 units
}

parameters {
  real<lower=0> sigma2theta;
  real<lower=0> sigma2eps;
  real mu;
}

transformed parameters {

  matrix[20,20] Sigma;
  matrix[20,20] SigmaM;
  vector[20] muvec;

  // construction of covariance matrix Sigma (Equation 5)
  // main diagonal
  for ( p in 1:P ){
    Sigma[p,p] = sigma2theta + sigma2eps;
  }
  // lower/upper triangle
  for (k in 2:P){
      for (l in 1:(k-1)) {
          Sigma[k,l] = sigma2theta;
          Sigma[l,k] = Sigma[k,l];
      }
  }

  // covariance matrix of means
  SigmaM = Sigma/J;

  // construction of mu vector
  for ( p in 1:P ){
    muvec[p] = mu;
  }
}

model {

  // distributional assumption of sample scatter matrix (Equation 3)
  S ~ wishart( J-1, Sigma );

  // distributional assumption of sample means (Equation 4)
  ybar ~ multi_normal( muvec, SigmaM );

  // prior distribution for mu
  mu ~ normal( 0, sqrt(10000) );

  // prior distribution for sigma2theta
  sigma2theta ~ inv_gamma(0.001,0.001);

  // prior distribution for sigma2eps
  sigma2eps ~ inv_gamma(0.001,0.001);
}
```

Appendix F. Stan Code (Classic Parametrization)

```
// Stan model code for the multi-level intercept-only model
// in the classic parametrization

data {
   int<lower=1> J; // number of level 2 units
   int<lower=1> P; // number of level 1 units
   matrix[P,J] D;  // data matrix
}

parameters {
   real<lower=0> sigma2theta;
   real<lower=0> sigma2eps;
   real mu;
   vector[J] theta;
}

model {
   // loop over level 2 units
   for (j in 1:J) {
      // loop over level 1 units
      for (p in 1:P) {
         // distributional assumption of observations (Equation 1)
         D[p,j] ~ normal( theta[j], sigma2eps );
      }
      // distributional assumption of the person parameters (Equation 2)
      theta[j] ~ normal( mu, sigma2theta );
   }

   // prior distribution for mu
   mu ~ normal( 0, sqrt(10000) );

   // prior distribution for sigma2theta
   sigma2theta ~ inv_gamma(0.001,0.001);

   // prior distribution for sigma2eps
   sigma2eps ~ inv_gamma(0.001,0.001);
}
```

Appendix G. Simulation Code

```
######################################
# R code for running the simulation #
######################################

## used R version: 4.0.5 (R Core Team, 2021)
# => get latest version: https://cran.r-project.org/

# used car version: 3.0-10
# => get latest car version: install.packages( "car" )
require(car)

# used rjags version: 4-10 (Plummer, 2019)
# => get latest rjags version: install.packages( "rjags" )
library(rjags)

# used rstan version: 2.21.2 (Stan Development Team, 2020)
# => get latest rstan version: install.packages( "rstan" )
library(rstan)

### function to generate data from the NULL model
##  v1 21.05.2021
#   mu.true        ... population mean
#   sigma2theta.true ... between person variance
#   sigma2eps.true  ... within person varianz
#   J               ... number of level 2 units (persons)
#   P               ... number of level 1 units (observations/time points)
#   value:             list of length 2; "y": P x J wide format matrix; "seed": seed
nullmodel_data_generation <- function( mu.true = 0, sigma2theta.true = 1, sigma2eps.true = 1, J = 1000, P = 20, seed=NULL ){

              # seed
              if( is.null( seed ) ) seed <- sample( 1:99999999, 1 )
              set.seed( seed )

              # data generation
              thetaj <- rnorm( J, mu.true, sqrt(sigma2theta.true) )
              y <- sapply( thetaj, function( mu ) rnorm( P, mu, sqrt(sigma2eps.true) ), simplify=TRUE )

              # return
              list( "y"=y, "seed"=seed )
}
```

```
### simulation conditions
ss   <- 1000                                                        ### sample size
nwarm<- c(150, 1000)                                                ### number of warmups
prior<- c(0.001, 0.01, 0.1, 1)                                      ### priors
iter <- 100000                                                      ### number of iterations
reps <- 1:1000                                                      ### number of replications
res  <- expand.grid ( nWarmups = nwarm, prior = prior, iter = iter, replication = reps, sampleSize = ss, stringsAsFactors = FALSE)
res  <- data.frame ( res, sequential_number = 1:nrow(res), stringsAsFactors = FALSE)

### define paths, need approx. 45 GB free space on hard disc
dir    <- "c:/stan_simulation"
outdir <- file.path(dir, "results")
workdir<- file.path(dir, "work")

### create folders if necessary
if(!dir.exists(outdir)) {dir.create(outdir)}
if(!dir.exists(workdir)) {dir.create(workdir)}

### jags and stan default model
jags   <- scan(url("https://figshare.com/ndownloader/files/31595090"), what="character",sep="\n",quiet=TRUE)
stan   <- scan(url("https://figshare.com/ndownloader/files/31595108"), what="character",sep="\n",quiet=TRUE)

### R functions to start jags/stan functions from R
# S ... sample scatter matrix
# y.bar ... sample mean column vector
# J ... number of level 2 units (persons)
# P ... number of level 1 units (observations/time points)
# n.iter ... number of iterations, default: 500
# warmup ... number of warmup iterations ("n.adapt"), default: 0
# bugs.file ... bugs/jags model syntax as plain text file
# inits ... list (chains) of lists (parameters) with init values; "auto" (default) sets inits to mu=0, prec.theta=1, prec.eps=1
# time.units ... character vector, time units, e.g. "secs" (default)
# value: list; first entry "res": data.frame (1st column "parameter", further columns: iterations)
#              second entry: named vector with run times for warmup, run, total
#              third entry "seeds": generated seed for running jags
nullmodel_jags <- function( S, y.bar, J, P, n.iter=500, warmup=0, bugs.file, inits="auto", time.units="secs" ){
              ## data list for JAGS
              data.list <- list()
              data.list <- c( data.list, list( "S"=S ) )
              data.list <- c( data.list, list( "y.bar"=y.bar ) )
              data.list <- c( data.list, list( "J"=J ) )
              data.list <- c( data.list, list( "P"=P ) )
              ## JAGS run parameters
              # thinning
              thin <- 1
              # starting values
              if ( inits %in% "auto" ){
                            # function to generate random starting values
                            generate.startval <- function(){
                               startval <- list()
```

```
                      # startval <- c( startval, list( "mu"    = rnorm(1,0,sqrt(10000)) ) )
                      startval <- c( startval, list( "mu"    = 0 ) )
                      # startval <- c( startval, list( "prec.theta" = rgamma(1,0.001,0.001)+10^(-20) ) )
                      startval <- c( startval, list( "prec.theta" = 1/1 ) )
                      # startval <- c( startval, list( "prec.eps" = rgamma(1,0.001,0.001)+10^(-20) ) )
                      startval <- c( startval, list( "prec.eps" = 1/1 ) )
                      return( startval )
                    }
                    # generate starting values for the chain
                    # set.seed( 1234 )
                    inits <- sapply( 1:1, function (chain.nr,l) l, generate.startval(), simplify=FALSE )
        }
        # set seeds for the chain
        seeds <- parallel.seeds( 'base::BaseRNG', 1 )
        inits <- mapply ( function(i,s) c(i,s), inits, seeds, SIMPLIFY=FALSE )
    ## initialize JAGS model
    # number of chains = 1 (code runs only for one chain)
    # number of adaptation iterations = 0 (no pre-sampling, just initialization)
    start.warmup <- Sys.time()
        ini <- jags.model ( file=bugs.file, data=data.list, n.chains=1, n.adapt=warmup,
                                      inits=inits, quiet=TRUE )
    # run time warmup
        runtime.warmup <- Sys.time() - start.warmup
        units( runtime.warmup )<- time.units
        runtime.warmup <- as.numeric( runtime.warmup )
    # run JAGS model
        track <- c("mu","sigma2.theta","sigma2.eps")
        start.run <- Sys.time()
        res <- jags.samples ( ini, variable.names=track, n.iter=n.iter, thin=thin,
                                         type='trace' , progress.bar = NULL )
    # run time main run
        runtime.run <- Sys.time() - start.run
        units( runtime.run )<- time.units
        runtime.run <- as.numeric( runtime.run )
        # total run time
        runtime <- runtime.warmup + runtime.run
        # named vector with runtimes
        runtimes <- c( runtime.warmup, runtime.run, runtime )
        names( runtimes ) <- c( "runtime.warmup", "runtime.run", "runtime" )
        # results
        dfr <- cbind( data.frame( "parameter"=names(res) ), do.call( "rbind", sapply( res, function(x) x[1,,1], simplify=FALSE ) ) )
        dfr <- dfr[ track, ]
        rownames( dfr ) <- seq( along=rownames( dfr ) )
        # return
        list( "samples"=dfr, "runtimes"=runtimes, "seed"=seeds )
}
```

```
### start simulation
res <- by(data = res, INDICES = res[,"sequential_number"], FUN = function ( bed) {
        dat  <- nullmodel_data_generation(J=bed[["sampleSize"]])
        d    <- dat[["y"]]
        y.bar <- matrix( rowMeans( d ), 20, 1 )                     ### mean vector (P = number of time points; D = matrix, persons x time
             points)
        ones <- matrix( 1, ncol(d), 1 )                             ### vector with 1 values
        S    <- ( d - y.bar %*% t(ones) ) %*% t( ( d - y.bar %*% t(ones) ) ) ### S matrix
     # first: jags. adapt priors in bugs file
        if(bed[,"prior"] != 0.001) {
           rows <- grep("dgamma", jags)
           jags[rows] <- gsub("0.001", bed[,"prior"], jags[rows])
           rows2<- grep("dnorm", jags)
           jags[rows2] <- gsub("10000", car::recode(bed[,"prior"],"0.01 = 2500; 0.1 = 100; 1 = 25"), jags[rows2])
        }
        dir.create(file.path(workdir, paste0(bed[,"sequential_number"], "_jags")))
        write(jags,file.path(workdir, paste0(bed[,"sequential_number"], "_jags"), "01b_multilevel_intonly_model.bugs"), sep="\n")
        fit  <- nullmodel_jags ( S=S, y.bar=y.bar, J=ncol(d), P=nrow(d), n.iter=bed[1,"iter"], warmup=bed[1,"nWarmups"], bugs.file = file.path(
             workdir, paste0(bed[,"sequential_number"], "_jags"), "01b_multilevel_intonly_model.bugs"), inits="auto", time.units="secs" )
        res.ja<- data.frame ( bed, software = "jags", runtime.translate = NA, runtime.compile = NA, runtime.samplingfunction = NA, t(fit[["
             runtimes"]]), stringsAsFactors = FALSE)
        colnames(fit[["samples"]]) <- car::recode(paste0("c", colnames(fit[["samples"]])), "'cparameter'='prm'")
        res.ja<- data.frame ( rbind(res.ja,res.ja,res.ja),fit[["samples"]], stringsAsFactors = FALSE)
     # second: jags (use identical data). adapt priors in bugs file
        if(bed[,"prior"] != 0.001) {
           rows <- grep("inv_gamma", stan)
           stan[rows] <- gsub("0.001", unique(bed[,"prior"]), stan[rows])
           rows2<- grep("normal\\( 0", stan)
           stan[rows2] <- gsub("10000", unique(car::recode(bed[,"prior"],"0.01 = 2500; 0.1 = 100; 1 = 25")), stan[rows2])
        }
        dir.create(file.path(workdir, unique(paste0(bed[,"sequential_number"], "_stan"))))
        write(stan,file.path(workdir, unique(paste0(bed[,"sequential_number"], "_stan")), "02b_multilevel_intonly_model.stan"), sep="\n")
        dlist <- list(J = ncol(d), P = nrow(d), S=S, ybar = as.vector(y.bar)) ### stan does not allow points in object names
        begin <- Sys.time()                                         ### first step: translate
        transl<- stanc(file = file.path(workdir, unique(paste0(bed[,"sequential_number"], "_stan")), "02b_multilevel_intonly_model.stan"),
             isystem = file.path(workdir, unique(paste0(bed[,"sequential_number"], "_stan"))))
        t1   <- Sys.time() - begin
        units(t1) <- "secs"
        begin2<- Sys.time()                                         ### second step: compile
        comp <- stan_model(stanc_ret = transl, auto_write = TRUE, isystem = file.path(workdir, unique(paste0(bed[,"sequential_number"], "_stan")
             )))
        t2   <- Sys.time() - begin2
        units(t2) <- "secs"
        begin3<- Sys.time()                                         ### third step: sampling
        fit  <- sampling(object = comp, data = dlist, iter = bed[1,"iter"] + bed[1,"nWarmups"], chains = 1, warmup = bed[1,"nWarmups"], init =
             list( list(mu=0, prectheta = 1, preceps = 1)), cores = 1)
        t3   <- Sys.time() - begin3
        units(t3) <- "secs"
        tmes <- attributes(fit@sim$samples[[1]])[["elapsed_time"]]
        res.st<- data.frame ( bed, software="stan", runtime.translate = as.numeric(t1), runtime.compile = as.numeric(t2), runtime.
```

```
        samplingfunction = as.numeric(t3), runtime.warmup = tmes[["warmup"]], runtime.run = tmes[["sample"]], runtime = tmes[["warmup"]]+
        tmes[["sample"]], stringsAsFactors = FALSE)
    chains<- as.data.frame(fit@sim[["samples"]][[1]])
    chainSelect    <- as.data.frame(t(chains[(bed[1,"nWarmups"]+1):nrow(chains),1:3]))
    colnames(chainSelect) <- paste0("c", as.character(1:bed[1,"iter"]))
    res.st<- data.frame ( rbind(res.st,res.st,res.st), chainSelect, stringsAsFactors = FALSE)
    res.st[,"prm"]    <- car::recode(rownames(chainSelect), "'sigma2theta'='sigma2.theta'; 'sigma2eps'='sigma2.eps'")
    resAll<- rbind(res.ja, res.st)
    save(resAll, file = file.path(outdir, paste0("results_",bed[,"sequential_number"])))
}
)
```

References

1. van de Schoot, R.; Winter, S.D.; Ryan, O.; Zondervan-Zwijnenburg, M.; Depaoli, S. A systematic review of Bayesian articles in psychology: The last 25 years. *Psychol. Methods* **2017**, *22*, 217–239. [CrossRef]
2. Zitzmann, S. A computationally more efficient and more accurate stepwise approach for correcting for sampling error and measurement error. *Multivar. Behav. Res.* **2018**, *53*, 612–632. [CrossRef]
3. Gilks, W.R.; Thomas, A.; Spiegelhalter, D.J. A language and program for complex Bayesian modelling. *Statistician* **1994**, *43*, 169–177. [CrossRef]
4. Lunn, D.; Jackson, C.; Best, N.; Thomas, A.; Spiegelhalter, D. *The BUGS Book*; CRC Press: Boca Raton, FL, USA, 2013.
5. Lunn, D.; Spiegelhalter, D.; Thomas, A.; Best, N. The BUGS project: Evolution, critique and future directions. *Stat. Med.* **2009**, *28*, 3049–3067. [CrossRef] [PubMed]
6. Lunn, D.J.; Thomas, A.; Best, N.; Spiegelhalter, D. WinBUGS—A Bayesian modelling framework: Concepts, structure, and extensibility. *Stat. Comput.* **2000**, *10*, 325–337. [CrossRef]
7. Spiegelhalter, D.; Thomas, A.; Best, N.; Lunn, D. OpenBUGS Version 3.2.3 User Manual. 2014. Available online: http://www.openbugs.net/w/Manuals (accessed on 15 May 2021).
8. Thomas, A.; O'Hara, R.; Ligges, U.; Sturtz, S. Making BUGS open. *R News* **2006**, *6*, 12–17. Available online: http://cran.r-project.org/doc/Rnews (accessed on 30 October 2021).
9. Plummer, M. JAGS: A program for analysis of Bayesian graphical models using Gibbs sampling. In Proceedings of the 3rd International Workshop on Distributed Statistical Computing (DSC 2003), Vienna, Austria, 20–22 March 2003.
10. de Valpine, P.; Turek, D.; Paciorek, C.J.; Anderson-Bergman, C.; Temple Lang, D.; Bodik, R. Programming with models: Writing statistical algorithms for general model structures with NIMBLE. *J. Comput. Graph. Stat.* **2017**, *26*, 403–413. [CrossRef]
11. Monnahan, C.C.; Thorson, J.T.; Branch, T.A. Faster estimation of Bayesian models in ecology using Hamiltonian Monte Carlo. *Methods Ecol. Evol.* **2017**, *8*, 339–348. [CrossRef]
12. Gelman, A.; Lee, D.; Guo, J. Stan: A probabilistic programming language for Bayesian inference and optimization. *J. Educ. Behav. Stat.* **2015**, *40*, 530–543. [CrossRef]
13. Hoffman, M.D.; Gelman, A. The No-U-turn sampler: Adaptively setting path lengths in Hamiltonian Monte Carlo. *J. Mach. Learn. Res.* **2014**, *15*, 1593–1623.
14. Neal, R. MCMC using Hamiltonian dynamics. In *Handbook of Markov Chain Monte Carlo*; Brooks, S., Gelman, A., Jones, G.L., Meng, X.-L., Eds.; Chapman and Hall/CRC: Boca Raton, FL, USA, 2011; pp. 116–162.
15. Nishio, M.; Arakawa, A. Performance of Hamiltonian Monte Carlo and No-U-Turn Sampler for estimating genetic parameters and breeding values. *Genet. Sel. Evol.* **2019**, *51*, 1–12. [CrossRef] [PubMed]
16. Betancourt, M. A conceptual introduction to Hamiltonian Monte Carlo. *arXiv* **2018**, arXiv:1701.02434.
17. Carpenter, B.; Gelman, A.; Hoffman, M.D.; Lee, D.; Goodrich, B.; Betancourt, M.; Brubaker, M.; Guo, J.; Li, P.; Riddell, A. Stan: A probabilistic programming language. *J. Stat. Softw.* **2017**, *76*, 1–32. [CrossRef]
18. Grant, R.L.; Furr, D.C.; Carpenter, B.; Gelman, A. Fitting Bayesian item response models in Stata and Stan. *Stata J.* **2017**, *17*, 343–357. [CrossRef]
19. Merkle, E.C.; Fitzsimmons, E.; Uanhoro, J.; Goodrich, B. Efficient Bayesian structural equation modeling in Stan. *arXiv* **2020**, arXiv:2008.07733.
20. Wingfeet. JAGS and Stan. 24 August 2014. Available online: https://www.r-bloggers.com/2014/08/jags-and-stan (accessed on 25 June 2021).
21. Bølstad, J. How Efficient is Stan Compared to JAGS? Conjugacy, Pooling, Centering, and Posterior Correlations. 2 January 2019. Available online: www.boelstad.net/post/stan_vs_jags_speed (accessed on 1 July 2021).
22. R Core Team. *R: A Language and Environment for Statistical Computing (Version 4.0.5) [Software]*; R Foundation for Statistical Computing: Vienna, Austria, 2021. Available online: https://www.r-project.org (accessed on 15 May 2021).
23. Plummer, M. JAGS (Version 4.3.0) [Software]. 2017. Available online: https://sourceforge.net/projects/mcmc-jags/files (accessed on 15 May 2021).

24. Plummer, M. rjags: Bayesian Graphical Models Using MCMC (Version 4–10) [Software]. 2019. Available online: https://cran.r-project.org/package=rjags (accessed on 15 May 2021).
25. Stan Development Team. Stan (Version 2.27) [Software]. 2021. Available online: https://mc-stan.org (accessed on 21 June 2021).
26. Stan Development Team. rstan: The R Interface to Stan (Version 2.21.2) [Software]. 2020. Available online: https://cran.r-project.org/package=rstan (accessed on 15 May 2021).
27. Hecht, M.; Gische, C.; Vogel, D.; Zitzmann, S. Integrating out nuisance parameters for computationally more efficient Bayesian estimation—An illustration and tutorial. *Struct. Equ. Model. A Multidiscip. J.* **2020**, *27*, 483–493. [CrossRef]
28. Hecht, M.; Zitzmann, S. A computationally more efficient Bayesian approach for estimating continuous-time models. *Struct. Equ. Model. A Multidiscip. J.* **2020**, *27*, 829–840. [CrossRef]
29. Goldstein, H. *Multilevel Statistical Models*; John Wiley & Sons: Hoboken, NJ, USA, 2011.
30. Gabry, J. Shinystan: Interactive Visual and Numerical Diagnostics and Posterior Analysis for Bayesian Models (Version 2.5.0) [Software]. 2018. Available online: http://cran.r-project.org/package=shinystan (accessed on 15 May 2021).
31. Turek, D.; de Valpine, P.; Paciorek, C.J. Efficient Markov chain Monte Carlo sampling for hierarchical hidden Markov models. *Environ. Ecol. Stat.* **2016**, *23*, 549–564. [CrossRef]
32. Zitzmann, S.; Hecht, M. Going beyond convergence in Bayesian estimation: Why precision matters too and how to assess it. *Struct. Equ. Model. A Multidiscip. J.* **2019**, *26*, 646–661. [CrossRef]
33. Zitzmann, S.; Weirich, S.; Hecht, M. Using the effective sample size as the stopping criterion in Markov chain Monte Carlo with the Bayes module in Mplus. *Psych* **2021**, *3*, 336–347. [CrossRef]
34. Nielsen, N.M.; Smink, W.A.C.; Fox, J.-P. Small and negative correlations among clustered observations: Limitations of the linear mixed effects model. *Behaviormetrika* **2021**, *48*, 51–77. [CrossRef]
35. Gelman, A.; Hill, J. *Data Analysis Using Regression and Multilevel/Hierarchical Models*; Cambridge University Press: Cambridge, UK, 2006.
36. Kruschke, J.K. *Doing Bayesian Data Analysis: A Tutorial Introduction with R and BUGS*, 2nd ed.; Academic Press: Cambridge, MA, USA, 2015.
37. Papaspiliopoulos, O.; Roberts, G.O.; Sköld, M. A General Framework for the Parametrization of Hierarchical Models. *Stat. Sci.* **2007**, *22*. [CrossRef]
38. Paganin, S.; Paciorek, C.J.; Wehrhahn, C.; Rodriguez, A.; Rabe-Hesketh, S.; de Valpine, P. Computational methods for Bayesian semiparametric Item Response Theory models. *arXiv* **2021**, arXiv:2101.11583.
39. Salvatier, J.; Wiecki, T.V.; Fonnesbeck, C. Probabilistic programming in Python using PyMC3. *PeerJ Comput. Sci.* **2016**, *2*. [CrossRef]
40. Statisticat, LLC. LaplacesDemon: Complete Environment for Bayesian Inference [Software]. 2021. Available online: https://web.archive.org/web/20150206004624/http://www.bayesian-inference.com/software (accessed on 22 August 2021).
41. Beraha, M.; Falco, D.; Guglielmi, A. JAGS, NIMBLE, Stan: A Detailed Comparison Among Bayesian MCMC Software. *arXiv* **2021**, arXiv:2107.09357.
42. De Valpine, P. Some Comparisons between NIMBLE, JAGS and Stan for a Couple of Examples from Gelman and Hill (2007). 8 August 2021. Available online: https://nature.berkeley.edu/~pdevalpine/MCMC_comparisons/some_ARM_comparisons/nimble_ARM_comparisons.html (accessed on 27 August 2021).
43. Ponisio, L.C.; Valpine, P.; Michaud, N.; Turek, D. One size does not fit all: Customizing MCMC methods for hierarchical models using NIMBLE. *Ecol. Evol.* **2020**, *10*, 2385–2416. [CrossRef]
44. Stan Development Team. Stan Reference Manual (Version 2.27). 2021. Available online: https://mc-stan.org/docs/2_27/reference-manual-2_27.pdf (accessed on 21 June 2021).
45. Stan Development Team. Stan User's Guide (Version 2.27). 2021. Available online: https://mc-stan.org/docs/2_27/stan-users-guide-2_27.pdf (accessed on 21 June 2021).