The Stata Journal (2017)
17, Number 2, pp. 343–357

Fitting Bayesian item response models in Stata and Stan

Robert L. Grant
BayesCamp
Croydon, UK
robert@bayescamp.com

Daniel C. Furr
University of California at Berkeley
Berkeley, CA

Bob Carpenter
Columbia University
New York, NY

Andrew Gelman
Columbia University
New York, NY

Abstract. Stata users have access to two easy-to-use implementations of Bayesian inference: Stata's native `bayesmh` command and StataStan, which calls the general Bayesian engine, Stan. We compare these implementations on two important models for education research: the Rasch model and the hierarchical Rasch model. StataStan fits a more general range of models than can be fit by `bayesmh` and uses a superior sampling algorithm, that is, Hamiltonian Monte Carlo using the no-U-turn sampler. Furthermore, StataStan can run in parallel on multiple CPU cores, regardless of the flavor of Stata. Given these advantages and given that Stan is open source and can be run directly from Stata do-files, we recommend that Stata users interested in Bayesian methods consider using StataStan.

Keywords: st0477, stan, windowsmonitor, StataStan, bayesmh, Bayesian

1 Introduction

Stata is widely used in the social sciences, economics, and biostatistics. In 2015, it became possible to routinely fit Bayesian models in Stata by using 1) Bayesian modeling commands introduced in Stata 14, which use the Metropolis–Hastings algorithm and Gibbs sampler, and 2) StataStan, an interface to the open-source Bayesian software, Stan (Grant 2015; Stan Development Team 2016). Previously, Bayesian methods were available in Stata only through user-written commands to interface with external software such as BUGS, JAGS, or MLwiN.

At the time of writing, the native Bayes implementation in Stata, `bayesmh`, allows a choice among 10 likelihood functions and 18 prior distributions. `bayesmh` is explicitly focused around regression models, although extensions to hierarchical (multilevel) models are possible with the inclusion of hyperpriors. Additionally, the user may write customized likelihood functions or customized posterior distributions.

Built for Bayesian inference, Stan is an open-source, collaboratively built software project that allows general continuous-parameter models, including all the models that can be fit in Stata's `bayesmh` and many others. Stan has been applied to a wide range of complex statistical models, including time series, imputation, mixture models, metaanalysis, cluster analysis, Gaussian processes, and item response theory. These extend beyond the current (Stata 14.2) capability of `bayesmh`. Stan can run from various data analysis environments such as Stata, R, Python, and Julia and also has a command-line interface (CmdStan). Stan uses Hamiltonian Monte Carlo (HMC) and the no-U-turn sampler (Hoffman and Gelman 2014) with the additional options of variational inference (Kucukelbir et al. 2015) and the L-BFGS optimization algorithm (Nocedal and Wright 2006). The advantages of HMC and the no-U-turn sampler in speed, stability with regard to starting values, and efficiency over Metropolis–Hastings and Gibbs have been described elsewhere (Neal 2011; Hoffman and Gelman 2014). As a result of the Hamiltonian dynamics, HMC is rotation invariant, which makes it well suited to highly correlated parameters. It is also not slowed down by nonconjugate models.

The languages used by these packages are notably different. In Stan, models are specified in a series of probability statements specifying prior distributions and likelihoods. `bayesmh` follows standard Stata syntax to give a compact specification of the most common regression and related models. Stan works by translating the user's model code into C++, then compiling and running the resulting executable file. Stan can run in parallel on multicore computers if the number of available cores was specified when installing CmdStan itself.

In this article, we compare `bayesmh` and StataStan on some item response models. These logistic regression (or Rasch) models are popular in education research and in political science, where they are called ideal-point models (Rasch 1960).

2 Models

We fit the models using data simulated as specified below. We checked that the `bayesmh` and StataStan implementations gave the same answer (modulo the inevitable Monte Carlo error of these stochastic algorithms), then compared the programs on speed and efficiency in terms of time per the number of effective independent samples.

The Rasch model can be written as

$$\Pr(y_{ip}=1|\theta_p,\delta_i) = \text{logit}^{-1}(\theta_p-\delta_i)$$
$$\theta_p \sim N(0,\sigma^2)$$

where $y_{ip}=1$ if person p responded to item i correctly and 0 otherwise, and $i,p \in \mathbb{N}$; $1 \leq i \leq I$; $1 \leq p \leq P$. The parameter θ_p represents the latent "ability" of person p, and δ_i is a parameter for item i. We considered a simple version of the model in which the abilities are modeled as exchangeable draws from a normal distribution with scale σ. We assigned a $N(0,10)$ prior distribution to δ_i and took two approaches to priors for σ. First, we matched the Rasch model example in the Stata 14 manual (see [BAYES] **bayesmh**), which uses an inverse-gamma prior for σ^2, which we do not recommend (Gelman 2006). Second, we used a preferred approach of uniform priors for σ, which is the Stan default if a prior is not specified. It is easy in StataStan to add a line of code to the model to include a different prior on σ or σ^2.

A natural hierarchical extension of the Rasch model adds a hyperprior for δ_i so that

$$\begin{aligned}\Pr(y_{ip}=1|\theta_p,\delta_i) &= \text{logit}^{-1}(\theta_p-\delta_i)\\ \theta_p &\sim N(0,\sigma^2)\\ \delta_i &\sim N(\mu,\tau^2)\end{aligned}$$

where μ is the model intercept. Persons and items are regarded as two sets of exchangeable draws.

3 Methods

We simulated data from the above model with 500 persons each answering 20 items. For true values of δ_i, we assigned equally spaced values from -1.5 to 1.5, and we set the true σ to 1.

We set up the Rasch and hierarchical Rasch models similarly, running four chains in series in Stan version 2.11 and Stata 14.1. We drew initial values for the chains from independent uniform distributions -1 to 1 on the location parameters $\mu^{(0)}$, $\delta^{(0)}$, and $\theta^{(0)}$ and drew uniform distributions from 0 to 2 on the scale parameters $\sigma^{(0)}$ and $\tau^{(0)}$. We assigned all δ_i's identical starting values for each chain and did the same for the θ_p's. The reason for this (admittedly unusual) choice is that this approach is much easier to use with `bayesmh`. We used the same starting values for both StataStan and `bayesmh` (and in the comparison described below, for JAGS). These item response models were not sensitive to starting values.

We ran 10 chains for 2,500 discarded warm-up iterations and 2,500 posterior draws each. For timing purposes, we ran all chains in serial, thus eliminating one of Stan's advantages—that it can automatically run multiple chains in parallel on a multicore machine regardless of the flavor of Stata. However, we made one comparison using parallel computation, which is described below. We provide the Stan programs and commands in the appendix. The options specified for `bayesmh` are nearly identical to those in the example provided in the Stata manual (see [BAYES] **bayesmh**). There is a difference in how the δ and θ parameters are sampled, which plays to the strengths of the different algorithms; Hamiltonian Monte Carlo is more efficient with distributions centered on or close to zero, regardless of correlation, while random walk Metropolis–Hastings in `bayesmh` is improved by using the random-effects option (see [BAYES] **bayesmh**). This feature, added in Stata 14.1, markedly improves effective sample size for models amenable to a random-effects parameterization. Other model forms will not benefit from it, so for comparison, we ran `bayesmh` both with and without random effects.

We monitored convergence for each parameter using the $\widehat{R}$ statistic, which is a rough estimate of the square root of the ratio of overall (across chains) posterior variance to within-chain posterior variance (Gelman et al. 2013). Values of $\widehat{R}$ near 1 indicate convergence, while greater values indicate nonconvergence. Values less than 1.1 are generally considered acceptable. The efficiency of the estimations is evaluated by the seconds per estimated effective sample size, $s/\widehat{n}_{\text{eff}}$ (Gelman et al. 2013). This reflects the fact that more highly autocorrelated chains of draws from the posterior distributions give less precise inference, equivalent to a smaller number of effectively independent samples that n_{eff} estimates. We used two versions of timings: an all-in time using the Stata command `timer` from the lines of the do-file above and below the `bayesmh` or `stan` command, as well as a simulation-only time obtained from the CmdStan output and from the value returned to `e(simtime)` by `bayesmh` (an undocumented return value). StataStan's all-in time includes compiling the model, and `bayesmh`'s all-in time includes internal model building before simulation can begin. To run multiple chains in StataStan, compilation is required only once, and if the data change but a model does not, a previously compiled executable file can be reused. The total time and simulation-only times represent opposite ends of a spectrum of performance. In real-life implementation, if there are many retained iterations compared with the warm-up iterations, and if compilation (in the case of StataStan) and model building (in the case of `bayesmh`) are not needed in every chain, total time will approach the simulation-only time.

To further investigate the efficiency of the software as models become more demanding, we carried out the same analyses on simulated data with 20 items and 100, 500, 1,000, 5,000, and 10,000 people. We compared StataStan 1.2.1 (calling CmdStan 2.11) with Stata 14.1's `bayesmh` command as above and also with the open-source software JAGS 4.0.0 (Plummer 2007) with the `rjags` package in R 3.2.3 and ran four chains in each instance. We compared $s/\widehat{n}_{\text{eff}}$ for the hyperparameters σ^2, μ, and τ^2 and for the worst parameter (lowest $\widehat{n}_{\text{eff}}$, reflecting the frequent need to run the software until all parameters are adequately estimated) in each model. We ran `bayesmh` both with and without the `exclude()` option on the θ's to examine the effect of reducing memory requirements. We also ran StataStan again with parallel chains for 20 items and 1,000 people to examine the increase in speed achieved with 4 CPU cores. All simulations were conducted on an "early 2015" MacBook Pro laptop running OS X 10.11.6 (El Capitan) with a 2.7 GHz Intel Core i5 4-core processor and 8 GB of 1867 MHz DDR3 RAM, with all networking turned off.

4 Results

For the Rasch model, we ran StataStan for 10 chains (in series) of 5,000 iterations (first half as warm-up) in 16.6 minutes; at that point, $\widehat{R}$ was less than 1.01 for all parameters. We ran `bayesmh` for 10 chains of the same length in 15.9 minutes; $\widehat{R}$ was less than 1.01 for all parameters. Convergence appears satisfactory for both. In figure 1, we compare values of time per effective independent sample for all the parameters in box plots between StataStan and `bayesmh`. Table 1 provides the same all-in timing statistics for the hyperparameters.

Table 1. Efficiency statistics for the hyperparameters in the two models

Model	Parameter	Stata 14.1 `bayesmh` $n_{\text{eff}}/\text{sec}$	StataStan $n_{\text{eff}}/\text{sec}$
Rasch	σ^2	1.44	8.99
Hierarchical Rasch	μ	3.80	1.22
Hierarchical Rasch	σ^2	1.63	5.62
Hierarchical Rasch	τ^2	3.28	2.66

Results for the hierarchical Rasch model parallel those for the Rasch model. Estimation with StataStan required 24.1 minutes for the same number of chains and iterations, and $\widehat{R}$ was less than 1.01 for all parameters. `bayesmh` ran for 16.6 minutes and yielded values of $\widehat{R}$ less than 1.01 for all parameters. Both estimations appear to have converged.

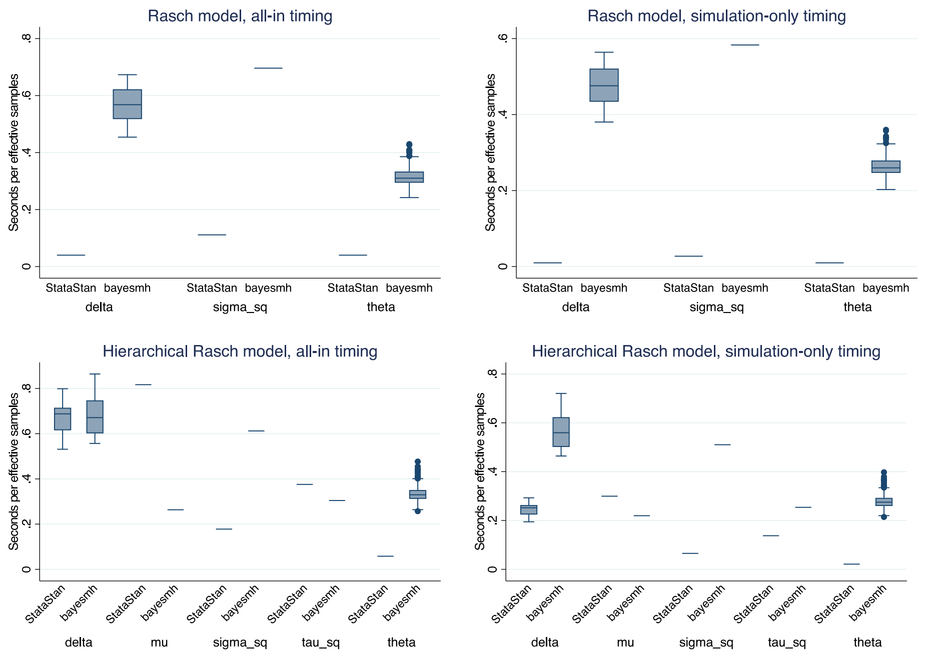

Figure 1. Box plots of seconds per effective independent sample for parameters in the Rasch model (top row of plots) and hierarchical Rasch model (bottom row), in each case fit to simulated data on 500 persons each answering 20 items. Left column shows total timing, including compilation and simulation; right column shows simulation time only. When a model is being fit multiple times, simulation-only timing is a more relevant comparison because the model needs to be compiled only once.

In terms of the total time from issuing the command to its completion, StataStan was more efficient for all parameters in the Rasch model; in the hierarchical Rasch model, it was more efficient for all θ's and σ^2, similar for the δ's, slightly less efficient for τ^2, and less efficient for μ. When we compared simulation-only time (not counting compilation, model building, or warm-up), StataStan's efficiency was improved, making all Rasch parameters even more favorable, and all hierarchical Rasch parameters except μ favor StataStan over `bayesmh`.

When we ran the models with the preferred StataStan priors and with sampling standard deviations rather than variances, results did not change much. Total computation time was somewhat faster at 11.1 minutes for the Rasch model and 22.6 minutes for the hierarchical Rasch model, but times per n_{eff} were very similar at 0.08 seconds for Rasch σ, 0.16 seconds for hierarchical Rasch σ, and 0.29 seconds for τ. However, the efficiency of μ improved to 0.49 seconds per n_{eff}.

Table 2. Efficiency statistics for hyperparameters in increasingly large models

			Total time		Simulation-only time	
			`bayesmh`	StataStan	`bayesmh`	StataStan
Model	Parameter	P	sec/n_{eff}	sec/n_{eff}	sec/n_{eff}	sec/n_{eff}
Rasch	σ^2	100	0.143	0.069	0.137	0.007
		500	0.536	0.105	0.502	0.025
		1,000	1.460	0.230	1.319	0.062
		5,000	9.333	1.649	6.404	0.576
		10,000	350.164	4.539	334.916	1.487
H. Rasch	μ	100	0.212	0.168	0.204	0.023
		500	0.211	0.760	0.197	0.287
		1,000	0.457	1.131	0.413	0.571
		5,000	2.682	22.025	1.847	11.331
		10,000	49.533	67.812	46.660	37.400
H. Rasch	σ^2	100	0.146	0.061	0.140	0.008
		500	0.595	0.177	0.558	0.067
		1,000	1.809	0.340	1.634	0.172
		5,000	11.941	4.508	8.225	2.319
		10,000	186.637	13.236	175.813	7.300
H. Rasch	τ^2	100	0.094	0.095	0.090	0.013
		500	0.350	0.385	0.328	0.145
		1,000	0.904	0.608	0.817	0.307
		5,000	5.145	8.237	3.544	4.237
		10,000	76.556	26.884	72.116	14.827

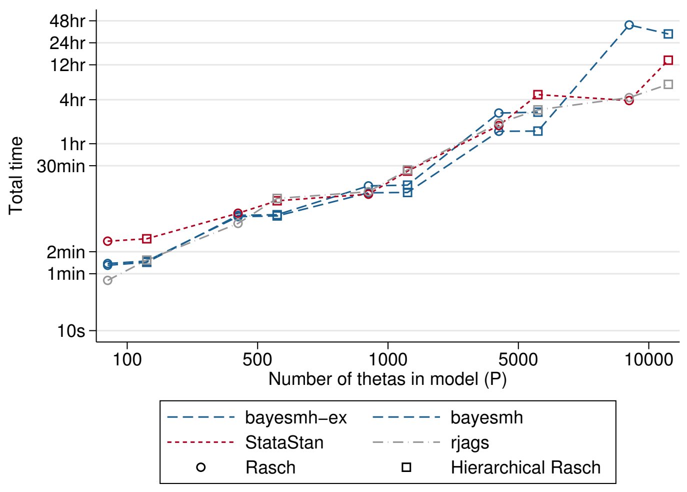

Figure 2. Total time per effective independent sample (worst efficiency across all parameters) in increasingly large Rasch and hierarchical Rasch models

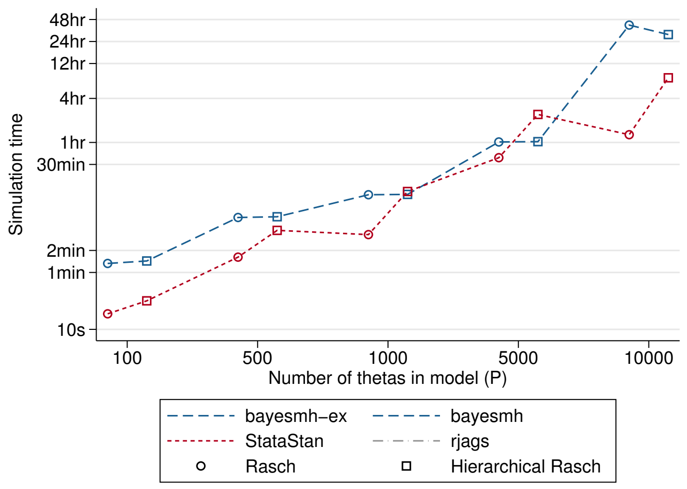

Figure 3. Simulation time per effective independent sample (worst efficiency across all parameters) in increasingly large Rasch and hierarchical Rasch models

In testing with increasingly large models, all three packages showed similar total execution times. StataStan had faster simulation-only time in all Rasch models (from 3% to 61% of the `bayesmh` times) and mixed results in hierarchical Rasch models (from 26% to 235%). The charts show the efficiency for the parameter with the lowest n_{eff} in each case, for total time (figure 2) and simulation-only time (figure 3). In these line charts, the form of `bayesmh` that uses the `exclude` option is denoted by `bayesmh-ex`; we found this had similar speed and efficiency to `bayesmh` without `exclude` and so did not assess it further in the most time-consuming models ($P = 10000$) or in terms of simulation-only time. JAGS does not provide simulation-only timings. In total time per n_{eff}, no one software option dominated, though from this limited simulation, it appeared that StataStan was more efficient in the smallest models and `bayesmh` was more efficient in the largest (figure 2). In simulation-only time per n_{eff}, StataStan was more efficient than `bayesmh`, up to 10 times more so in most models and sizes of P. However, in some cases, they were similar, with `bayesmh` being slightly better (figure 3).

StataStan consistently had the better efficiency (s/n_{eff}) for σ^2 in both Rasch and hierarchical Rasch models but mixed results for τ^2, although four out of five models favored StataStan in simulation-only time (table 2). In the Rasch model, StataStan was 2.1 to 77.1 times faster than `bayesmh` to achieve the same n_{eff} for σ^2 in total time and 11.1 to 225.2 times faster in simulation-only time. In the hierarchical Rasch model, StataStan was 2.4 to 14.1 times faster for σ^2 in total time and 3.5 to 24.1 times faster in simulation-only time. StataStan was 0.6 to 2.8 times faster for τ^2 in total time and 0.8 to 6.9 times faster in simulation-only time. The μ hyperparameter in the hierarchical Rasch models was more efficiently sampled by `bayesmh` at most values of P, with StataStan being 0.1 to 1.3 times faster in total time and 0.2 to 8.9 times faster in simulation-only time (table 2). All models, with all software, could be fit with the same laptop computer without running out of memory.

The random-effects option in `bayesmh` provided a considerable improvement in both effective sample size and speed. When we ran the $I = 20, P = 100$ models without random effects, total time was 206 seconds for Rasch and 211 seconds for hierarchical Rasch, while simulation-only times were 200 and 204 seconds, respectively, which is about 2.5 times slower than the same model with random effects. The time per effective independent sample was considerably increased. In the Rasch model, it rose from 0.143 to 69 seconds for σ^2. In the hierarchical Rasch model, it rose from 0.146 to 30 seconds for σ^2, from 0.212 to 53 seconds for μ, and from 0.094 to 23 seconds for τ^2.

A further consideration is the speed-up obtained by running StataStan chains in parallel even without Stata/MP. We found that the Rasch model with $I = 20$ and $P = 1000$ had total time 383 seconds running in parallel compared with 734 seconds running in series and simulation-only time of 78 seconds compared with 198 seconds. The hierarchical Rasch model of the same size had total time 850 seconds compared with 1,520 seconds and simulation-only time of 303 seconds compared with 768 seconds. This would make parallel StataStan on a quad-core computer roughly twice as efficient as serial StataStan, while `bayesmh` will not run parallel chains without Stata/MP.

5 Discussion

We found that most of the Rasch models we compared were more efficiently sampled by StataStan than **bayesmh** and that this was more favorable to StataStan because the fixed overhead of compiling the model into an executable file was outgrown by the simulation. This suggests that longer chains of draws from the posterior distribution, precompiled models, and parallel chains will all favor StataStan, and the total time comparisons here represent a worst-case scenario for StataStan. We would expect the results we found to apply to generalized linear mixed models, given that these include the Rasch models as a special case (Rijmen et al. 2003; Zheng and Rabe-Hesketh 2007). In practice, the adaptive Markov chain Monte Carlo algorithm featured in **bayesmh** (Stata 14.1) also has a number of features that improve its performance notably over JAGS or Stata 14.0. We found that the same models without the **reffects** option took 200 to 500 times longer to achieve the same effective sample size on **bayesmh**, which users should keep in mind when considering models outside the Rasch family and without random effects.

StataStan provides a simple interface, operating by writing specified variables (as vectors), matrices, and scalars from Stata to a text file and calling the command-line implementation of Stan. The user can specify a wide variety of priors, and the algorithm is less sensitive to the prior than that used in **bayesmh**. In these Rasch models, we found it simple and more intuitive to sample the hyperparameters as standard deviations rather than variances or precisions, and we used uniform priors as the Stan default without any impact on efficiency. We give the alternative programs in the appendix. Progress is displayed inside Stata (even under Windows), and there is the option to write the Stan model inside a comment block in the Stata do-file. Results can then be read back into Stata for diagnostics, generating other values of interest, or saving in `.dta` format. StataStan can be installed from Statistical Software Components by typing

```
ssc install stan
```

Windows users should also type

```
ssc install windowsmonitor
```

In conclusion, we find StataStan to be generally faster than **bayesmh**, which is no surprise given Stan's advanced algorithms and efficient autodifferentiation code. Given that Stan is open source, offers a wider range of models than **bayesmh**, and can be run directly from Stata using StataStan, we recommend that Stata users interested in Bayesian methods consider StataStan for Bayesian modeling, especially for more complicated models.

6 Acknowledgment

We thank the Institute of Education Sciences for partial support of this work.

7 References

Gelman, A. 2006. Prior distributions for variance parameters in hierarchical models. *Bayesian Analysis* 1: 515–534.

Gelman, A., J. B. Carlin, H. S. Stern, D. B. Dunson, A. Vehtari, and D. B. Rubin, eds. 2013. *Bayesian Data Analysis*. 3rd ed. Boca Raton, FL: CRC Press.

Grant, R. L. 2015. StataStan. https://github.com/stan-dev/statastan.

Hoffman, M. D., and A. Gelman. 2014. The no-U-turn sampler: Adaptively setting path lengths in Hamiltonian Monte Carlo. *Journal of Machine Learning Research* 15: 1593–1623.

Kucukelbir, A., R. Ranganath, A. Gelman, and D. M. Blei. 2015. Automatic variational inference in Stan. ArXiv Working Paper No. arXiv:1506.03431. https://arxiv.org/abs/1506.03431.

Neal, R. M. 2011. MCMC using Hamiltonian dynamics. In *Handbook of Markov Chain Monte Carlo*, ed. S. Brooks, A. Gelman, G. L. Jones, and X.-L. Meng, 113–162. Boca Raton, FL: Chapman & Hall/CRC.

Nocedal, J., and S. J. Wright. 2006. *Numerical Optimization*. 2nd ed. Berlin: Springer.

Plummer, M. 2007. JAGS. http://mcmc-jags.sourceforge.net/.

Rasch, G. 1960. *Probabilistic Models for Some Intelligence and Attainment Tests*. Copenhagen: Danmarks Pædagogiske Institut.

Rijmen, F., F. Tuerlinckx, P. De Boeck, and P. Kuppens. 2003. A nonlinear mixed model framework for item response theory. *Psychological Methods* 8: 185–205.

Stan Development Team. 2016. Stan Modeling Language: User's Guide and Reference Manual. http://mc-stan.org/documentation/.

Zheng, X., and S. Rabe-Hesketh. 2007. Estimating parameters of dichotomous and ordinal item response models with gllamm. *Stata Journal* 7: 313–333.

About the authors

Robert Grant is a statistician who runs a training and consultancy startup called BayesCamp. He was senior lecturer in health and social care statistics at Kingston University and St George's, University of London, and previously worked on clinical quality and safety data for the Royal College of Physicians and evidence-based guidelines for the National Institute for Clinical Excellence. Besides Bayesian modeling, he specializes in data visualization.

Daniel Furr is a graduate student in the Graduate School of Education at the University of California, Berkeley. He is the developer of `edstan`, an R package for Bayesian item response theory modeling using Stan, and the author of several related case studies. He specializes in item response theory and predictive methods.

Bob Carpenter is a research scientist in computational statistics at Columbia University. He designed the Stan probabilistic programming language and is one of the Stan core developers. He was professor of computational linguistics (Carnegie Mellon University) and an industrial researcher and programmer in speech recognition and natural language processing (Bell Labs, SpeechWorks, LingPipe) and has written two books on programming language theory and linguistics.

Andrew Gelman is a professor of statistics and professor of political science at Columbia University and the author of several books, including *Bayesian Data Analysis*.

Appendix

Here is the code for the models, starting with the Rasch Stan program, which matches `bayesmh`:

```
data {
  int<lower=1> N;              // number of observations in the dataset
  int<lower=1> I;              // number of items
  int<lower=1> P;              // number of people
  int<lower=1, upper=I> ii[N]; // variable indexing the items
  int<lower=1, upper=P> pp[N]; // variable indexing the people
  int<lower=0, upper=1> y[N];  // binary outcome variable
}
parameters {
  real<lower=0> sigma_sq; // variance of the thetas (random intercepts for people)
  vector[I] delta_unit;   // normalized deltas
  vector[P] theta_unit;   // normalized thetas
}
transformed parameters {
  real<lower=0> sigma;
  sigma = sqrt(sigma_sq); // SD of the theta random intercepts
}
model {
  vector[I] delta;
  vector[P] theta;
  theta_unit ~ normal(0, 1);     // prior for normalized thetas
  delta_unit ~ normal(0, 1);     // prior for normalized deltas
  sigma_sq ~ inv_gamma(1, 1);    // prior for variance of thetas
  theta = theta_unit * sigma;    // convert normalized thetas to thetas (mean 0)
  delta = delta_unit * sqrt(10); // convert normalized deltas to deltas (mean 0)
  y ~ bernoulli_logit(theta[pp] - delta[ii]); // likelihood
}
```

This is our preferred Stan program:

```
data {
  int<lower=1> N;              // number of observations in the dataset
  int<lower=1> I;              // number of items
  int<lower=1> P;              // number of people
  int<lower=1, upper=I> ii[N]; // variable indexing the items
  int<lower=1, upper=P> pp[N]; // variable indexing the people
  int<lower=0, upper=1> y[N];  // binary outcome variable
}
parameters {
  real<lower=0> sigma;  // SD of the thetas (random intercepts for people)
  vector[I] delta_unit; // normalized deltas
  vector[P] theta_unit; // normalized thetas
}
model {
  vector[I] delta;
  vector[P] theta;
  theta_unit ~ normal(0, 1);     // prior for normalized thetas
  delta_unit ~ normal(0, 1);     // prior for normalized deltas
  theta = theta_unit * sigma;    // convert normalized thetas to thetas (mean 0)
  delta = delta_unit * sqrt(10); // convert normalized deltas to deltas (mean 0)
  y ~ bernoulli_logit(theta[pp] - delta[ii]); // likelihood
}
```

Here is the Stata call for the Rasch model:

```
bayesmh y=({theta:}-{delta:}), likelihood(logit)           ///
   redefine(delta:i.item) redefine(theta:i.person)         ///
   prior({theta:i.person}, normal(0, {sigmasq}))           ///
   prior({delta:i.item}, normal(0, 10))                    ///
   prior({sigmasq}, igamma(1, 1))                          ///
   mcmcsize(`mcmcsize´) burnin(`burnin´)                   ///
   notable saving(`draws´, replace) dots                   ///
   initial({delta:i.item} `=el(inits`jj´, `c´, 1)´         ///
      {theta:i.person} `=el(inits`jj´, `c´, 2)´            ///
      {sigmasq} `=el(inits`jj´, `c´, 3)´)                  ///
   block({sigmasq})
```

And here is the JAGS code for the Rasch model:

```
model {
  for (i in 1:I) {
    delta[i] ~ dunif(-1e6, 1e6)
  }
  inv_sigma_sq ~ dgamma(1,1)
  sigma <- pow(inv_sigma_sq, -0.5)
  for (p in 1:P) {
    theta[p] ~ dnorm(0, inv_sigma_sq)
  }
  for (n in 1:N) {
    logit(inv_logit_eta[n]) <- theta[pp[n]] - delta[ii[n]]
    y[n] ~ dbern(inv_logit_eta[n])
  }
}
```

Here is the hierarchical Rasch model in Stan, matching `bayesmh`:

```
data {
  int<lower=1> N;              // number of observations in the dataset
  int<lower=1> I;              // number of items
  int<lower=1> P;              // number of people
  int<lower=1, upper=I> ii[N]; // variable indexing the items
  int<lower=1, upper=P> pp[N]; // variable indexing the people
  int<lower=0, upper=1> y[N];  // binary outcome variable
}
parameters {
  real<lower=0> sigma_sq; // variance of the thetas (random intercepts for people)
  real<lower=0> tau_sq;   // variance of the deltas (random intercepts for items)
  real mu;                // mean of the deltas
  vector[I] delta_unit;   // normalized deltas
  vector[P] theta_unit;   // normalized thetas
}
transformed parameters {
  real<lower=0> sigma;
  real<lower=0> tau;
  sigma = sqrt(sigma_sq); // SD of the theta random intercepts
  tau = sqrt(tau_sq);     // SD of the delta random intercepts
}
model {
  vector[I] delta;
  vector[P] theta;
  theta_unit ~ normal(0, 1);       // prior for normalized thetas
  delta_unit ~ normal(0, 1);       // prior for normalized deltas
  mu ~ normal(0, sqrt(10));        // prior for the mean of the deltas
  sigma_sq ~ inv_gamma(1, 1);
  tau_sq ~ inv_gamma(1, 1);
  theta = theta_unit * sigma;      // convert normalized thetas to thetas (mean 0)
  delta = mu + (delta_unit * tau); // convert normalized deltas to deltas (mean mu)
  y ~ bernoulli_logit(theta[pp] - delta[ii]); // likelihood
}
```

This is our preferred Stan model:

```
data {
  int<lower=1> N;              // number of observations in the dataset
  int<lower=1> I;              // number of items
  int<lower=1> P;              // number of people
  int<lower=1, upper=I> ii[N]; // variable indexing the items
  int<lower=1, upper=P> pp[N]; // variable indexing the people
  int<lower=0, upper=1> y[N];  // binary outcome variable
}
parameters {
  real<lower=0> sigma;  // SD of the thetas (random intercepts for people)
  real<lower=0> tau;    // SD of the deltas (random intercepts for items)
  real mu;              // mean of the deltas
  vector[I] delta_unit; // normalized deltas
  vector[P] theta_unit; // normalized thetas
}
model {
  vector[I] delta;
  vector[P] theta;
  theta_unit ~ normal(0, 1);       // prior for normalized thetas
  delta_unit ~ normal(0, 1);       // prior for normalized deltas
  mu ~ normal(0, sqrt(10));        // prior for the mean of the deltas
  theta = theta_unit * sigma;      // convert normalized thetas to thetas (mean 0)
  delta = mu + (delta_unit * tau); // convert normalized deltas to deltas (mean mu)
  y ~ bernoulli_logit(theta[pp] - delta[ii]); // likelihood
}
```

Here is the Stata call for the hierarchical Rasch model:

```
bayesmh y=({theta:}-{delta:}),likelihood(logit)           ///
   redefine(delta:i.item) redefine(theta:i.person)        ///
   prior({theta:i.person}, normal(0, {sigmasq}))          ///
   prior({delta:i.item}, normal({mu}, {tausq}))           ///
   prior({mu}, normal(0, 10))                             ///
   prior({sigmasq} {tausq}, igamma(1, 1))                 ///
   block({sigmasq} {tausq} {mu}, split)                   ///
   initial({delta:i.item} `=el(inits`jj´, 1, 1)´          ///
      {theta:i.person} `=el(inits`jj´, 1, 2)´             ///
      {sigmasq} `=el(inits`jj´, 1, 3)´                    ///
      {tausq} `=el(inits`jj´, 1, 4)´                      ///
      {mu} `=el(inits`jj´, 1, 5)´)                        ///
   mcmcsize(`mcmcsize´) burnin(`burnin´)                  ///
   saving(`draws´, replace) dots
```

And here is the JAGS code for the hierarchical Rasch model:

```
model {
  inv_sigma_sq ~ dgamma(1,1)
  sigma <- pow(inv_sigma_sq, -0.5)
  for (p in 1:P) {
    theta[p] ~ dnorm(0, inv_sigma_sq)
  }
  inv_tau_sq ~ dgamma(1,1)
  tau <- pow(inv_tau_sq, -0.5)
  for (i in 1:I) {
    delta[i] ~ dnorm(0, inv_tau_sq)
  }
  mu ~ dunif(-1e6, 1e6)
  for (n in 1:N) {
    logit(inv_logit_eta[n]) <- mu + theta[pp[n]] - delta[ii[n]]
    y[n] ~ dbern(inv_logit_eta[n])
  }
}
```