The Stata Journal (2017)
17, Number 2, pp. 330–342

Introducing the StataStan interface for fast, complex Bayesian modeling using Stan

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

Bob Carpenter
Columbia University
New York, NY

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

Andrew Gelman
Columbia University
New York, NY

Abstract. In this article, we present StataStan, an interface that allows simulation-based Bayesian inference in Stata via calls to Stan, the flexible, open-source Bayesian inference engine. Stan is written in C++, and Stata users can use the commands `stan` and `windowsmonitor` to run Stan programs from within Stata. We provide a brief overview of Bayesian algorithms, details of the commands available from Statistical Software Components, considerations for users who are new to Stan, and a simple example. Stan uses a different algorithm than `bayesmh`, BUGS, JAGS, SAS, and MLwiN. This algorithm provides considerable improvements in efficiency and speed. In a companion article, we give an extended comparison of StataStan and `bayesmh` in the context of item response theory models.

Keywords: st0476, stan, windowsmonitor, StataStan, Bayesian, bayesmh, interface, shell commands, Stan

1 Introduction

Stata users have long been able to seamlessly access other software specializing in Bayesian analysis, thanks to Stata users' abilities to write arbitrary information to ASCII text files and send commands to the operating system. This allowed for commands such as `runmlwin` (Leckie and Charlton 2013) and `wb` (Thompson 2017) to send data and code to MLwiN and WinBUGS, respectively, then collect the results and display them inside Stata for further calculation and graphing. Since 2015, when Stata 14 was released, Stata users have been able to use a native implementation of Bayesian simulation algorithms by using the `bayesmh` command. However, `bayesmh` is limited to regressionlike models where a dependent variable has a specified likelihood conditional on some function of independent variables, and can allow only certain prior distributions. It cannot, for example, fit structural equation models, Gaussian processes, or spatial correlation models.

The day after Stata 14's release, StataStan was published online (Stan Development Team 2016b). StataStan is an umbrella term for all commands and programs necessary to interface with Stan from Stata. It can be installed from Statistical Software Components by typing

```
ssc install stan
```

and Windows users should also install `windowsmonitor` by typing

```
ssc install windowsmonitor
```

Stan is an open-source, collaboratively built software project that implements an algorithm (Hamiltonian Monte Carlo) for Bayesian modeling that is faster and more stable than the algorithms (random walk Metropolis–Hastings and the Gibbs sampler) implemented in BUGS, JAGS, SAS, MLwiN, and `bayesmh`. Stan has been applied to a wide range of complex statistical models, including time series, imputation, mixture models, meta-analysis, cluster analysis, Gaussian processes, and item response theory. These extend beyond the current (Stata 14.2) capability of `bayesmh`, which is explicitly for regression. In our companion article (Grant et al. 2017), we describe the functionality of Stan and advantages of its algorithm. In this article, we give a brief overview of Hamiltonian Monte Carlo in intuitive terms, set out the syntax of the commands, and present a worked example.

2 Hamiltonian Monte Carlo

All Bayesian methods make estimates and inference by evaluating posterior distributions, combinations of likelihood based on data, and a model with prior distributions representing uncertainty about parameters of the model before the data were known. Different practitioners take the prior to mean different concepts, in the same way that "uncertainty" and "probability" are not rigorously defined concepts despite decades of hard work by statisticians and philosophers of science. Regardless of the interpretation, Bayesian methods differ from frequentist methods in that they allow probability statements to be made about any unknown value, not just those that represent eternally replicable random sampling.

Textbook examples often start with algebraically tractable posterior distributions, but in practice, this is generally either infeasible or too time consuming and prone to human error to be worthwhile. Instead, software allows the analyst to run one or more Markov chains of pseudorandom values that converge to a stationary distribution equivalent to draws from the joint posterior distribution of all parameters of interest. From a large enough number of these draws, estimation and inference can be done empirically. The older algorithms, random walk Metropolis–Hastings, and the Gibbs sampler take random steps through parameter space and accept or reject the new location based on its posterior probability.

This can work well under some circumstances but under others can require large numbers of draws before they accurately represent the posterior distribution (conver-

gence). Problems like this commonly arise when parameters are correlated (like, for example, how the intercept and slope of a bivariate linear regression are correlated with only a small amount of data); when priors are not ideal matches for the likelihood (a subtle topic beyond the scope of this article but discussed in Bayesian textbooks [Gelman et al. 2013]); or when initial values are poor guesses. Hamiltonian Monte Carlo addresses these issues by using Hamilton's equations of motion with periodic random impulses (Neal 2011). Exploration of the posterior probability is then analogous to a particle moving in a force field (picture a beachball rolling in the hollow between sand dunes, with occasional random kicks—gravity is the force providing the Hamiltonian motion); as the joint posterior distribution guides movement to the region of highest posterior probability, the problems of sampling using random steps disappear. Even chains with poor initial values can still reveal the whole posterior distribution relatively quickly (Neal 2011). A computer requires computationally expensive numerical integration and differentiation to perform this imitation of life, but the lifting of the problems associated with random walk Metropolis–Hastings and Gibbs more than compensates for this. The no-U-turn sampler is the algorithm implemented in Stan (Hoffman and Gelman 2014), which automatically tunes the parameters of Hamiltonian Monte Carlo, achieving nearly optimal integration time in recent tests using CmdStan (Betancourt 2016).

In the companion article, we present a comparison of the efficiency of StataStan alongside `bayesmh` for an item response model (Grant et al. 2017).

3 The stan and windowsmonitor commands

3.1 Objectives and development

Building on the history of linking Stata to WinBUGS (Thompson 2017), we sought to provide one command that would dispatch a specified Stan model code along with data. Because Stata can easily issue operating system commands, we use this to run the command-line implementation of Stan (CmdStan) and display summary results inside Stata. This is the approach also taken by the Stan interfaces from MATLAB and Julia. CmdStan has to be installed before using StataStan, but this is relatively straightforward with instructions on the Stan website, http://mc-stan.org.

We believe that Stata users who are becoming familiar with Bayesian techniques will find StataStan a flexible, stable, and fast tool. Also, people who already use Stata and Stan separately will find it helpful to keep everything in one workflow, because most people find it easier to work with one piece of software than to switch among them (and find it easier to maintain quality control). This allows data processing, simple analysis, complex modeling, graphics, and report writing all in one place.

One unexpected problem we encountered was that Windows does not make its standard output on the command line available in such a way that Stata can display it in the results window until the external program has finished execution. In the case of complex Bayesian models that can take hours to run, this would be unacceptable. Therefore, we wrote a small companion program called `windowsmonitor` that displays command-line

output close to real time. `windowsmonitor` may provide a useful alternative to `shell` and `winexec` in other settings too.

3.2 The stan command for Stata

`stan` specifies what data are to be sent to CmdStan, with options controlling its settings and additional requirements such as sampling diagnostics or posterior modes. Data are passed to CmdStan in a text file, and outputs are returned similarly. These files are temporarily created in the CmdStan directory, then moved to the working directory. There is an option to retain all files. Otherwise, unnecessary files are deleted afterward. Users should be mindful that any existing files in these locations with these names may be overwritten. A model has to be stored in its own file with extension `.stan`, and we discuss different ways to achieve this below.

Syntax of stan

`stan` *varlist* [*if*] [*in*] [`,` `datafile(`*filename*`)` `modelfile(`*filename*`)` `inline`
`thisfile(`*filename*`)` `rerun` `initsfile(`*filename*`)` `load` `diagnose`
`outputfile(`*filename*`)` `chainfile(`*filename*`)` `mode` `modesfile(`*filename*`)`
`winlogfile(`*filename*`)` `seed(`*integer*`)` `warmup(`*integer*`)` `iter(`*integer*`)`
`thin(`*integer*`)` `chains(`*integer*`)` `skipmissing` `matrices(`*string*`)` `globals(`*string*`)`
`keepfiles` `stepsize(`*integer*`)` `stepsizejitter(`*integer*`)`]

Options

`datafile(`*filename*`)` specifies the name (and path if desired) of a text file where `stan` will write the data on its way to Stan. This is done in the format used by R/S-Plus and BUGS. For example, with the sample 1978 Automobile dataset,

```
stan mpg, ...
```

would write

```
mpg=c(...)
```

The default is `datafile(statastan_data.R)`.

`modelfile(`*filename*`)` specifies the name (and path if desired) of a text file containing the Stan model. The file must have the extension `.stan`. If this file already exists, the model is read from there, or the model can be written into the file using one of the methods detailed below under *Specifying the Stan model*. The default is `modelfile(statastan_model.stan)`.

`inline` instructs Stata to read the `.stan` model from a comment block inside the do-file (see below under *Specifying the Stan model* for further discussion of `modelfile()`, `inline`, and `thisfile()`).

`thisfile(`*filename*`)` specifies the name (and path if required) of the current do-file; this is an option if `inline` has been specified (see below under *Specifying the Stan model* for further discussion of `modelfile()`, `inline`, and `thisfile()`).

`rerun` uses the existing executable file with the same name as `modelfile()` (in Windows, it will have the extension `.exe`). This should exist in the working directory. Be aware it will be copied into `cmdstandir` (see below), deleting any existing file of that name.

`initsfile(`*filename*`)` specifies the name of a text file in R/S-Plus format containing initial values. Because Stan is far less sensitive to initial values than software using older algorithms, we do not presently provide any mechanism like the `datafile()` option to write this file from inside Stata.

`load` instructs Stata to read in the resulting draws as its current dataset.

`diagnose` runs Stan's diagnostics and displays them after sampling to examine whether the algorithm has run successfully.

`outputfile(`*filename*`)` provides the name (and path if required) for the text file into which CmdStan will write its outputs. The default is `outputfile(output.csv)`.

`chainfile(`*filename*`)` provides the name for a comma-separated values format file that will contain the draws from CmdStan; this is the same as `outputfile()`, but extra information is removed so it can be read into Stata using import delimited. The default is `chainfile(statastan_chains.csv)`.

`mode` runs Stan's optimization to find posterior modes and displays the results after sampling; it will also write the output into `modesfile()` (see below).

`modesfile(`*filename*`)` provides the name of a text file to hold output from CmdStan's estimation of modes. The default is `modesfile(modes.csv)`.

`winlogfile(`*filename*`)` provides the name of a temporary file to hold Windows output (see `windowsmonitor`); `windowsmonitor` will display the output in Stata's Results window, so there is no need to change the name of the temporary file from the default, which is `winlog.txt`.

`seed(`*integer*`)` provides an integer pseudorandom-number generator seed for Stan.

`warmup(`*integer*`)` specifies the number of warmup draws, which are discarded from the output and summaries. The default is `warmup(1000)`.

`iter(`*integer*`)` specifies the number of iterations (draws) to retain after `warmup()`. The default is `iter(1000)`.

`thin(`*integer*`)` specifies how much Stan thins draws; if `thin()` is set to n, Stan will retain one out of every n draws in output files and use the thinned draws for summaries. The default is `thin(1)` (no thinning).

`chains(`*integer*`)` determines how many chains to run, in parallel if possible (regardless of the Stata flavor installed).

`skipmissing` removes missing data observations (on a cell-by-cell basis inside each column) before sending data to Stan. This is relevant if you want to send a series of vectors of different sizes by making these appear as "variables" in your Stata data. This could be useful in the context of multilevel models with smaller vectors of cluster-level data. It is not a natural way to think of Stata data, so it should be used with caution because it will apply to all the variables in *varlist*.

`matrices(`*string*`)` provides a list of matrices to send to Stan or if set to `all`, it will send all current matrices. These are written into `datafile()` as two-dimensional arrays.

`globals(`*string*`)` provides a list of global macros to send to Stan, or `all` to send all current global macros. These are written into `datafile()` as scalars. The user should not write a string value, because this will probably cause an error from CmdStan.

`keepfiles` instructs `stan` to keep all files produced along the way; otherwise, the model file, C++ file, executable file, chains file, and (if produced) modes file will be retained in the working directory.

`stepsize(`*integer*`)` sets the stepsize for Hamiltonian Monte Carlo. The default is `stepsize(1)` (see the Stan manual [Stan Development Team 2016b] for more details).

`stepsizejitter(`*integer*`)` sets the stepsize jitter for Hamiltonian Monte Carlo. The default is `stepsizejitter(0)` (see the Stan manual [Stan Development Team 2016b] for more details).

4 Specifying the Stan model

You can specify the Stan model in at least three ways. First, you can write a `.stan` file externally, for example, in a text editor, then name it with the `modelfile()` option. This has the disadvantage that updating the analysis may require synchronized changes in the do-file and model file. However, we recommend this method as the starting point for new StataStan users because it avoids any bugs in writing and reading text files and allows you to begin immediately using examples from the Stan manual and website. Second, you can include the code inside a comment block in the do-file. If you use the `inline` and `thisfile()` options, Stata will read the text contents of `thisfile()` and identify the comment block that begins (on the line following the `/*` symbol) with the word `data`, as seen below:

```
/*
data {
  int<lower=0> N;
  int<lower=0,upper=1> y[N];
}
parameters {
  real<lower=0,upper=1> theta;
}
model {
  theta ~ beta(1,1);
  y ~ bernoulli(theta);
}
*/
```

Stata will then write the contents of the block to the `.stan` file specified in `modelfile()`.

Third, you can include the model code in the do-file as a series of strings in a `foreach` loop, which writes each line to the `modelfile()`. This has the advantage that all Stata and Stan code is in one file, but does not rely on naming or finding the do-file.

At present, the `inline` approach (option two above) does not accommodate multiple blocks of code, but we intend to add this capability.

4.1 The windowsmonitor program

`windowsmonitor` is a wrapper extending the ability of `shell`. It will be called by `stan` under Windows only; it will return an error message if it is used in Mac or Linux computers. It intercepts the `stdout` stream (text displayed on the screen for command line programs) and prints it inside Stata. It does this by diverting `stdout` to a text file, checking that file every two seconds for new content, and displaying that in Stata if it finds any. This continues until it receives a message that it is finished (in the form of a final line of output, `Finished!`), which is added automatically. The user should avoid using `windowsmonitor` to carry out any task that could write one `Finished!` line for any reason, because this will terminate the display inside Stata prematurely. However, if this is unavoidable, it is relatively simple to amend the signal word `Finished!` in the source code. `windowsmonitor` creates a file called `wmbatch.bat`. If this survives execution, it can safely be deleted later.

Syntax of windowsmonitor

`windowsmonitor, command(`*string*`)` [`waitsecs(`*integer*`)` `winlogfile(`*filename*`)`]

Options

`command(`*string*`)` contains the Windows command-line code to be sent for execution. `command()` is required.

`waitsecs(`*integer*`)` specifies the number of seconds to wait for output to appear before giving up. The default is `waitsecs(20)`.

`winlogfile(`*filename*`)` specifies the file (and path if desired) to store the output in; by default, a Stata `tempfile` will be used, so there is nothing to be gained from specifying a `tempfile` macro here. The default is `winlogfile(winlog.txt)`.

5 Considerations for newcomers to Stan

Newcomers are strongly advised to work through some of the examples in the Stan manual before attempting serious applications. The Stan user must specify the type (such as integer or real number) as data or parameters. This allows Stan to make efficient calculations and helps with checking for inadvertent errors at compile time. Stan will translate the model to C++, which is itself a "typed" language. For the most part, the Stata user need not be concerned with this other than with the obvious choice when writing the Stan code. However, one potential pitfall may arise when reading in data from nonnative file formats into Stata and sending it with `stan`. Floating-point precision means that what the person reads may not match what the computer stores, and this may lead to a "type mismatch" error message from CmdStan.

The statistics reported by CmdStan and hence displayed by `stan` are the mean of draws from the posterior; the Monte Carlo standard error representing the uncertainty in the results arising from a finite number of draws; the standard deviation; the 5th, 50th, and 95th centiles of the draws; the number of effective independent samples (`N_Eff`, which accounts for autocorrelation in the chains) and number of effective independent samples obtained per second (`N_Eff/s`); and a measure of convergence (`R_hat`). The calculation of these measures is set out in Gelman et al. (2013). `N_Eff` and `R_hat` are best assessed across multiple chains, so we advise users run at least four chains as a general rule. `stan` can run parallel chains on multicore computers, even if Stata/MP is not installed, so most modern laptops can run four chains simultaneously. In the authors' experience, this runs in about half the time of serial chains.

Beyond these reported statistics, the value of loading the draws from the posterior distributions is that custom-derived values can be calculated and summarized by the user inside Stata to provide decision-theoretic outputs. To give an example from health economics, we can load a meta-analysis from Stan providing inference on the effectiveness of alternative drugs into Stata and combine it with constant costs to derive a new cost-effectiveness variable, which allows probability statements about whether the cost effectiveness exceeds a willingness-to-pay threshold. Another important benefit of working with the posterior draws is that the covariance structure among the parameters is preserved, while the tabulated summaries provide only marginal inferences.

Another consideration is that the number of available CPU cores needs to be specified when installing CmdStan itself, and the StataStan `chains()` option can parallelize only up to this number (Stan Development Team 2016a).

Stan model code allows for vectorized statements such as

```
y ~ bernoulli(theta);
```

instead of

```
for (n in 1:N) { y[n] ~ bernoulli(theta); }
```

Both can be used in Stan, but the vectorized version is generally faster in execution.

6 Example

All the models set out in the Stan manual and website can be fit directly using StataStan, including many models that are not possible in `bayesmh`. We can use StataStan for a simple example to estimate the probability of success θ in a Bernoulli process,

$$\Pr(y_i) = \theta, \quad 1 \leq i \leq 10, \quad i \in \mathbb{N}$$

when we have 10 outcomes: 8 failures and 2 successes. We will apply a flat prior distribution over $[0, 1]$, either by explicitly specifying it or by omitting it because Stan uses uniform priors as default, provided that bounds on the parameter have been specified. The corresponding `bayesmh` command is

```
bayesmh y, likelihood(dbernoulli({theta})) prior({theta},beta(1, 1))
```

The Stan code for this example is contained in the examples folder inside CmdStan.

```
data {
  int<lower=0> N;
  int<lower=0,upper=1> y[N];
}
parameters {
  real<lower=0,upper=1> theta;
}
model {
  theta ~ beta(1,1);
  y ~ bernoulli(theta);
}
```

The code is arranged in blocks of data, parameters, and model. Other block types can also be included, described fully in the Stan manual. Each object in the model, whether data or parameter, must be declared with its type and constraints before it can be used. Like BUGS and JAGS, the assignment operator $<-$ is used to calculate a value and store it in the object named on the left-hand side, while the `~` operator has two functions. In the line

```
theta ~ beta(1,1);
```

we are specifying a prior distribution (because theta is already declared as a parameter), and in the line

```
y[i] ~ bernoulli(theta);
```

we are incrementing the log probability by the likelihood contribution of one observation according to the Bernoulli probability given the current estimate of theta.

Having specified this model, we can make the data,

```
clear
set obs 10
generate y=0
replace y=1 in 2
replace y=1 in 10
```

and then call `stan`:

```
quietly count
global N=r(N)
global cmdstandir "/path_to/CmdStan"
stan y, modelfile("bernoulli.stan") cmd("$cmdstandir") globals("N")
```

StataStan first displays its own version number and then the CmdStan version installed in `cmdstandir`. The first output to be displayed concerns translating the model to C++, then compiling the resulting code. Compiling can be time consuming but does not have to be done again unless the model changes. If StataStan finds CmdStan successfully, and CmdStan is properly installed, this line will appear followed by some output that users can ignore:

```
--- Translating Stan model to C++ code ---
```

Next, a block of code will appear, starting with this line and comprising the command to the g++ compiler program (which is installed as part of CmdStan):

```
--- Linking C++ model ---
```

After compilation, we will see some settings for CmdStan, including the number of samples to retain and the number to use as warm-up:

```
method = sample (Default)
  sample
    num_samples = 1000 (Default)
    num_warmup = 1000 (Default)
```

We then see the iterations appear, followed by a total time to do the sampling:

```
Iteration: 1800 / 2000 [ 90%]  (Sampling)
Iteration: 1900 / 2000 [ 95%]  (Sampling)
Iteration: 2000 / 2000 [100%]  (Sampling)
Elapsed Time: 0.017155 seconds (Warm-up)
              0.024054 seconds (Sampling)
              0.041209 seconds (Total)

Inference for Stan model: bernoulli_model
4 chains: each with iter=(1000,1000,1000,1000);
  warmup=(0,0,0,0); thin=(1,1,1,1);
  4000 iterations saved.
```

```
Warmup took (0.017, 0.017, 0.017, 0.016) seconds,
  0.067 seconds total
Sampling took (0.024, 0.032, 0.031, 0.031) seconds,
  0.12 seconds total
```

This is followed by a summary of the parameters:

```
        Mean     MCSE   StdDev     5%   50%   95%    N_Eff  N_Eff/s    R_hat
theta   0.25  2.3e-03  1.2e-01  0.076  0.24  0.46     2784    23545  1.0e+00

Samples were drawn using hmc with nuts.
For each parameter, N_Eff is a crude measure of effective
  sample size, and R_hat is the potential scale reduction
  factor on split chains (at convergence, R_hat=1).
```

This shows us that we ran 4 chains and retained 1,000 samples from each, but because of autocorrelation, this was equivalent to 2,784 independent samples (23,545 independent samples per second). The posterior mean for θ was 0.25 (pulled upward from the maximum likelihood estimate by the flat prior and the small dataset). If `mode` is specified, we will then see the posterior mode,

```
Log-probability at maximum: -5.004020214080811
```

Parameter	Posterior Mode
theta	.200004

which is directly comparable (with a flat prior) with the maximum likelihood estimate, 0.2.

If we specify `diagnose`, we will see corresponding output; see the Stan manual for details on this.

```
TEST GRADIENT MODE
 Log probability=-7.10591
 param idx      value      model  finite diff          error
         0  -0.557247   -1.37022     -1.37022  -1.66588e-010
```

Finally, if we specify `load`, we will see a Stata-generated summary, including the 95% credible interval:

variable	N	mean	sd	se(mean)
theta	1000	.2485084	.1121162	.0035454

variable	min	p1	p5	p25
theta	.019246	.0477189	.0814933	.1628

variable	p50	p75	p95	p99
theta	.244064	.3222845	.4458295	.5513045

```
95% CI for theta: .0656607497483492 to .4934002541005615
```

This is similar to the approximate confidence interval:

```
. cii proportions 10 2, wilson
```

				Wilson	
Variable	Obs	Proportion	Std. Err.	[95% Conf.	Interval]
	10	.2	.1264911	.0566822	.5098375

We see the data replaced with variables called `theta` (which contains draws for the parameter of that name), `lp__`, `accept_stat__`, `stepsize__`, `treedepth__`, `n_leapfrog__`, and `n_divergent__`, all of which are created by CmdStan to track progress of the algorithm and can be safely deleted unless needed for methodological investigations. The `theta` variable, containing the draws from the posterior, can then be used for graphics or further inference.

7 Conclusion

Stan continues to develop rapidly, with one major project being the inclusion of Riemann manifold Hamiltonian Monte Carlo, which will provide further significant improvements in speed and stability (Girolami and Calderhead 2011). StataStan can readily track this by adding new options that are passed to future versions of CmdStan.

Stan and all its interfaces have been made possible by enthusiastic contributions from developers around the world, coordinated by a core team. We encourage all interested Stata users to visit http://mc-stan.org and become involved there through reporting issues and suggesting improvements (Stan Development Team 2016b).

8 Acknowledgments

We thank the Institute of Education Sciences for partial support of this work. We are also grateful to users who tested StataStan and provided feedback and to John Thompson of the University of Leicester and Charles Opondo of the University of Oxford for suggesting ways of inline model specification.

9 References

Betancourt, M. 2016. Identifying the optimal integration time in Hamiltonian Monte Carlo. ArXiv Working Paper No. arXiv:1601.00225. https://arxiv.org/abs/1601.00225.

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.

Girolami, M., and B. Calderhead. 2011. Riemann manifold Langevin and Hamiltonian Monte Carlo methods. *Journal of the Royal Statistical Society, Series B* 73: 123–214.

Grant, R. L., D. C. Furr, B. Carpenter, and A. Gelman. 2017. Fitting Bayesian item response models in Stata and Stan. *Stata Journal* 17: 343–357.

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.

Leckie, G., and C. Charlton. 2013. runmlwin—A program to run the MLwiN multilevel modelling software from within Stata. *Journal of Statistical Software* 52(11): 1–40.

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.

Stan Development Team. 2016a. CmdStan Interface: User's Guide. https://github.com/stan-dev/cmdstan/releases/download/v2.14.0/cmdstan-guide-2.14.0.pdf.

———. 2016b. Stan Modeling Language: User's Guide and Reference Manual, Version 2.14.0. https://github.com/stan-dev/stan/releases/download/v2.14.0/stan-reference-2.14.0.pdf.

Thompson, J. 2017. WinBUGS from Stata. http://www2.le.ac.uk/departments/health-sciences/research/gen-epi/Progs/winbugs-from-stata.

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*.