

Stan: A probabilistic programming language for Bayesian inference and optimization*

Andrew Gelman[†]

Daniel Lee[†]

Jiqiang Guo[‡]

6 Aug 2015

Abstract

Stan is a free and open-source C++ program that performs Bayesian inference or optimization for arbitrary user-specified models and can be called from the command line, R, Python, Matlab, or Julia, and has great promise for fitting large and complex statistical models in many areas of application. We discuss Stan from users' and developers' perspectives and illustrate with a simple but nontrivial nonlinear regression example.

1. What is Stan? Users' perspective

Stan, named after Stanislaw Ulam, a mathematician who was one of the developers of the Monte Carlo method in the 1940s (Metropolis & Ulam, 1949), is a C++ program to perform Bayesian inference. The code is open-source and is available at <http://mc-stan.org/> along with instructions and a 500-page user manual. Stan 1.0 was released in 2012 and, as of this writing, is currently in version 2.6.

To use Stan, a user writes a Stan program that directly computes the log-posterior density. This code is then compiled and run along with data. The result is a set of posterior simulations of the parameters in the model (or a point estimate, if Stan is set to optimize). Stan can be run from the command line, R, Python, Matlab, or Julia, and its output can be saved, printed, or graphed in R using shinyStan.

Stan can be thought of as similar to BUGS (Lunn, Thomas, Best, & Spiegelhalter, 2000) and Jags (Plummer, 2003) in that it allows a user to write a Bayesian model in a convenient language whose code looks like statistics notation. Or Stan can be thought of as an alternative to programming a sampler or optimizer oneself. Stan uses the no-U-turn sampler (Hoffman & Gelman, 2014), an adaptive variant of Hamiltonian Monte Carlo (Neal, 2011), which itself is a generalization of the familiar Metropolis algorithm, performing multiple steps per iteration to move more efficiently through the posterior distribution.

From the user's perspective, the main limitation of Stan is that it does not allow inference for discrete parameters. Stan allows discrete *data* and discrete-data *models* such as logistic regressions, but it cannot perform inference for discrete *unknowns*. Many statistical applications with discrete parameters are mixture models and can be rewritten more efficiently in terms of continuous parameters as discussed in the chapter on latent discrete parameters in the Stan manual (Stan Development Team, 2015), but there are some examples where this sort of re-expression does not exist.

In addition, there are models that are slow to run or do not converge in Stan, a problem it shares with any statistical inference engine. It is well known that optimization and inference can never be performed in complete generality; there will always be optimization or inference problems that are beyond our current capacities. It is the goal of Stan to push these capacities forward.

*For the *Journal of Educational and Behavioral Statistics*. We thank the Institute for Education Sciences, the National Science Foundation, the Sloan Foundation, and the Office of Naval Research for partial support of this work.

[†]Department of Statistics and Department of Political Science, Columbia University, New York.

[‡]NPD Group, Port Washington, New York.

2. Example: Fitting a simple nonlinear model

We demonstrate Stan by fitting the model, $y = a_1 e^{-b_1 x} + a_2 e^{-b_2 x}$, to simulated data. This sort of curve arises in the analysis of physical systems (for example, a two-compartment model in pharmacology), and it is easy to write down and understand but can be difficult to fit when data are sparse or when the two coefficients b_1 and b_2 are close to each other.

Our first step is to formulate this as a stochastic rather than deterministic model. The simplest approach would be to add an independent error term, but in this sort of problem the data are typically restricted to be positive, so we shall use multiplicative errors with a non-normal distribution, thus, $y_i = (a_1 e^{-b_1 x_i} + a_2 e^{-b_2 x_i}) \cdot \epsilon_i$, for $i = 1, \dots, n$, with $\log \epsilon_i \sim N(0, \sigma^2)$.

The model needs to be constrained to separately identify the two components; here, we shall restrict b_1 to be less than b_2 .

The corresponding Stan model, starting with noninformative priors on the parameters a, b, σ , looks like this:

```
data {
  int N;
  vector[N] x;
  vector[N] y;
}
parameters {
  vector[2] log_a;
  ordered[2] log_b;
  real<lower=0> sigma;
}
transformed parameters {
  vector<lower=0>[2] a;
  vector<lower=0>[2] b;
  a <- exp(log_a);
  b <- exp(log_b);
}
model {
  vector[N] ypred;
  ypred <- a[1]*exp(-b[1]*x) + a[2]*exp(-b[2]*x);
  y ~ lognormal(log(ypred), sigma);
}
```

We explain this program, block by block:

- The *data* block declares the data that must be input into the Stan program. In this case, we will input the data via a call from R. This particular model requires an integer and two (real-valued) vectors; Stan also supports matrices, ordered vectors, simplexes (a vector constrained to be nonnegative and sum to 1), covariance matrices, correlation matrices, and Cholesky factors. In addition, you can define arrays of arbitrary dimension of any of these objects; for example, `matrix[N,K] z[J1,J2,J3]`; would define a $J_1 \times J_2 \times J_3$ array of matrices, each of dimension $N \times K$. These sorts of arrays can be useful for multilevel modeling.
- The *parameters* block introduces all the unknown quantities that Stan will estimate. In this case, we have decided for convenience to parameterize the model coefficients a and b in terms of their logarithms. The parameter vector `log_a` is unrestricted, whereas `log_b` is constrained to be “ordered” (i.e., increasing) so that the two terms of the model can be identified. Finally,

`sigma`, the scale of the error term, needs to be estimated, and it is constrained to be nonnegative. Stan also allows upper bounds; for example, `real<lower=0,upper=1> c;` would define a parameter constrained to be between 0 and 1.

- *Transformed parameters* are functions of data and parameters. In this example, we define `a` and `b` by exponentiating `log_a` and `log_b`. This allows the model to be written in a more readable way.
- The *model* block is where the log-posterior density is computed. In this particular example we are using a flat prior (that is, uniform on the four parameters, $\log_a[1], \log_a[2], \log_b[1], \log_b[2]$, and so we just supply the likelihood. The last line of the model block is a vector calculation and has the effect of adding N terms to the log-likelihood, with each term being the log-density of the lognormal distribution. We define the predicted value `ypred` in its own separate line of code to make the model more readable.

We can try this out this program by simulating fake data in R and then fitting the Stan model. Here is the R code:

```
# Set up the true parameter values
a <- c(.8, 1)
b <- c(2, .1)
sigma <- .2

# Simulate data
x <- (1:1000)/100
N <- length(x)
ypred <- a[1]*exp(-b[1]*x) + a[2]*exp(-b[2]*x)
y <- ypred*exp(rnorm(N, 0, sigma))

# Fit the model
library("rstan")
fit <- stan("exponentials.stan", data=list(N=N, x=x, y=y), iter=1000, chains=4)
print(fit, pars=c("a", "b", "sigma"))
```

We have just run Stan for 4 chains of 1000 iterations each. The computations are automatically performed in parallel on our laptop which has 4 processors. In the call to `print`, we have specified that we want to see inferences for the parameters `a`, `b`, and `sigma`. The default would be to show all the parameters, but in this case we have no particular interest in seeing `log_a` and `log_b`.

Here is the R output:

```
Inference for Stan model: exponentials.
4 chains, each with iter=1000; warmup=500; thin=1;
post-warmup draws per chain=500, total post-warmup draws=2000.
```

	mean	se_mean	sd	2.5%	25%	50%	75%	97.5%	n_eff	Rhat
a[1]	1.00	0.00	0.03	0.95	0.99	1.00	1.02	1.05	494	1
a[2]	0.70	0.00	0.08	0.56	0.65	0.69	0.75	0.87	620	1
b[1]	0.10	0.00	0.00	0.09	0.10	0.10	0.10	0.11	532	1
b[2]	1.71	0.02	0.34	1.15	1.48	1.67	1.90	2.49	498	1
sigma	0.19	0.00	0.00	0.19	0.19	0.19	0.20	0.20	952	1

Samples were drawn using NUTS(diag_e) at Wed Mar 25 16:07:15 2015.

For each parameter, `n_eff` is a crude measure of effective sample size, and `Rhat` is the potential scale reduction factor on split chains (at convergence, `Rhat=1`).

We can quickly summarize the inference for any parameter by its posterior mean and standard deviation (*not* the column labeled `se_mean` which gives the Monte Carlo standard error of the mean), or else using the posterior quantiles. In any case, in this well-behaved example, Stan has recovered the true parameters to a precision given by their inferential uncertainty. The only change is that the labeling of the two terms has switched, but that is to be expected given that the model cannot tell them apart and we have arbitrarily constrained b_1 to be less than b_2 .

3. Continuing the example: Regularizing using a prior distribution

The above example is simple, perhaps deceptively so. The sum-of-exponentials model is notoriously ill-conditioned, and it is common for there to be no stable solution.

Indeed, here is what happens when we re-run Stan with the identical code as above but change the true coefficients from (2, 0.1) to (0.2, 0.1) in the code used to simulate the data:

	mean	se_mean	sd	2.5%	25%	50%	75%	97.5%	n_eff	Rhat
a[1]	1.33e+00	0.54	0.77	0.00	1.28	1.77e+00	1.79e+00	1.82e+00	2	44.23
a[2]	2.46e+294	Inf	Inf	0.00	0.00	0.00e+00	1.77e+00	2.66e+188	2000	NaN
b[1]	1.00e-01	0.04	0.06	0.00	0.10	1.30e-01	1.30e-01	1.40e-01	2	33.64
b[2]	3.09e+305	Inf	Inf	0.13	0.50	1.15e+109	4.77e+212	5.28e+298	2000	NaN
sigma	2.00e-01	0.00	0.00	0.19	0.19	2.00e-01	2.00e-01	2.00e-01	65	1.06

Bad news!

Actually, this is far from the worst possible outcome, as it is obvious from these results that the algorithm has not converged to a sensible solution: in this case, the coefficients β are estimated at infinity, which corresponds to a zero time scale. Running the chains for longer does not help, and running an optimizer to get a maximum likelihood estimate also fails catastrophically.

In problems like this where there is no stable solution, the Bayesian answer is to add prior information. In this case we make the assumption that the parameters a and b are likely to be not far from a unit scale. We set this up as a regularization or soft constraint by adding the following two lines in the `model` block of the Stan program:

```
log_a ~ normal(0, 1); // I've hardcoded the prior mean and sd of log_a and log_b.
log_b ~ normal(0, 1); // If model is well-scaled, these priors are weakly informative.
```

When we run this new program (using the same R script as before, just pointing to the file with the updated Stan code), the results are much cleaner:

	mean	se_mean	sd	2.5%	25%	50%	75%	97.5%	n_eff	Rhat
a[1]	1.56	0.09	0.32	0.65	1.52	1.72	1.75	1.79	13	1.25
a[2]	0.32	0.08	0.28	0.06	0.14	0.22	0.37	1.13	13	1.20
b[1]	0.13	0.00	0.01	0.10	0.12	0.13	0.13	0.14	22	1.14
b[2]	1.94	0.20	2.29	0.14	0.22	1.26	3.00	7.34	127	1.05
sigma	0.20	0.00	0.00	0.19	0.19	0.20	0.20	0.21	656	1.00

Recall that the true parameter values in this simulation are $a = (0, 8, 1)$, $b = (0.2, 0.1)$, but the coefficients b have been constrained to be increasing, so the two terms have switched their labels.

A careful examination of the quantiles reveals that the posterior distribution for b_2 is highly skewed, implying that the data are consistent with high values of this parameter, or equivalently, a nearly zero time scale for this term of the model. The parameters in this model are only weakly identified, as we can see further from the scatterplots of posterior simulations in Figure 1.

In other simulations of this model we have encountered much more skewed posteriors, even with the strong unit normal prior. It is generally good practice to simulate fake data multiple times and check that one's statistical procedure reliably reconstructs the assumed parameter values.

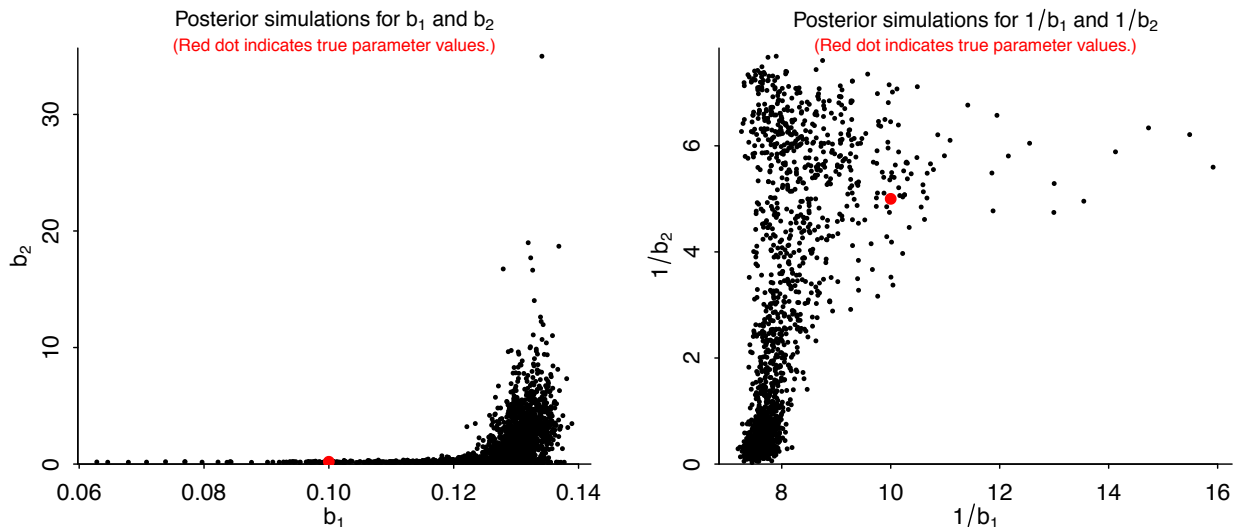


Figure 1: For each graph, the black dots show 2000 posterior simulation draws of the parameters b , obtained by Stan fitting the model, $y = (a_1 e^{-b_1 x} + a_2 e^{-b_2 x}) \cdot \text{error}$ to simulated data. The red dot on each graph shows the true parameter values that were used to create the simulated data. We display inferences for the key parameters b_1 and b_2 , which are only weakly identified in this particular dataset. We graph the parameters first as is, and then their reciprocals, which corresponds to the time scales of the two exponentially decaying terms in the model.

The user can experiment with the prior distribution, just as he or she can experiment with any features of the model. For example, if we change the unit normal priors to unit t_4 distributions—these have wider tails and will thus admit more extreme parameter values if they are consistent with the data—the posteriors, unsurprisingly, become broader. In the simulation above, the 95% posterior interval for b_2 changes from $[0.14, 7.3]$, as shown in the above output, to $[0.14, 122]$. Such a wide uncertainty reflects the weakness of these data to estimate the parameters in this nonlinear model, and motivates the use of a strong prior.

It is also easy to extend the model in Stan in other ways, for example by constructing a hierarchical specification with data from multiple groups in which the parameters can vary by group.

4. How does Stan work? Developers' perspective

Stan has several components:

- A flexible modeling language where the role of a Stan program is to **compute a log posterior density**. Stan programs are translated to templated C++ for efficient computation.
- An **inference engine which performs Hamiltonian Monte Carlo** using the no-U-turn sampler (Hoffman & Gelman, 2014) to get approximate simulations from the posterior distribution which is defined by the Stan program and by the data supplied to it from the calling program.
- In addition, **a L-BFGS optimizer** (Byrd, Lu, Nocedal, & Zhu, 1994) which iterates to find a (local) maximum of the objective function (in Bayesian terms, a posterior mode) and can also be used as a stand-alone optimizer.
- The sampler and the optimizer both require gradients. **Stan computes gradients using reverse-mode automatic differentiation** (Griewank & Walther, 2008), a procedure in which Stan's

compiler takes the function defined by the called Stan program and analytically computes its derivative using an efficient procedure which can be much faster than numerical differentiation, especially when the number of parameters is large.

- Routines to monitor the convergence of parallel chains and compute inferences and effective sample sizes (Gelman et al., 2013).
- Wrappers to call Stan from R, Python, Stata, Matlab, Julia, and the command line, and to graph the results. Where possible these calling functions automatically run Stan in parallel; for example, on a laptop with four processors, Stan can be set up to run one chain on each.

Stan is open source, lives on GitHub, and is being developed by several groups of people. At the center is the core team, which currently numbers 17 people. The core team, or subsets of it, meets weekly via videoconference, directs Stan research and development, and coordinates Stan projects. The developers group currently includes 46 people who exchange ideas on listserv which is open to the public. Various features of Stan including a model for correlation matrices (Lewandowski, Kurowicka, & Joe, 2009), higher default acceptance rate for adaptation, and vectorization of all the univariate distributions came from the developers list. Stan also has a users group where people can post and answer questions. There are currently 1100 people on this list and it averages 15 messages per day.

The users group is diverse, including researchers in astronomy, ecology, physics, medicine, political science, education, economics, epidemiology, population dynamics, pharmacokinetics, and many others.

The core team is actively involved in responding to user questions, partly because the ultimate goal of the project is to serve research, and partly because user problems typically represent gaps in performance or, at the very least, gaps in communication.

5. Comparison to other software

Stan was motivated by the desire to solve problems that could not be solved in reasonable time (user programming time plus run time) using other packages. There are also problems where Stan does not work so well.

In comparing Stan to other software options, we consider several criteria:

1. Flexibility, that is, being able to fit the desired model.
2. Ease of use; user programming time.
3. Run time.
4. Scalability as dataset and model grow larger.

We go through each of these criteria for the following software options:

Bugs and Jags. Bugs is an automatic Bayesian inference engine written in Component Pascal written and developed largely from 1989–2004, that uses Gibbs sampling and other algorithms to draw posterior samples from arbitrary user-specified graphical models (Lunn, Spiegelhalter, Thomas, & Best, 2009). Jags is a variant of Bugs, written in C++ and developed from 2007–2013 (Plummer, 2003).

Compared to Bugs and Jags, Stan is more flexible in that its modeling language is more general—with the major exception that those other packages can handle discrete parameters and Stan cannot,

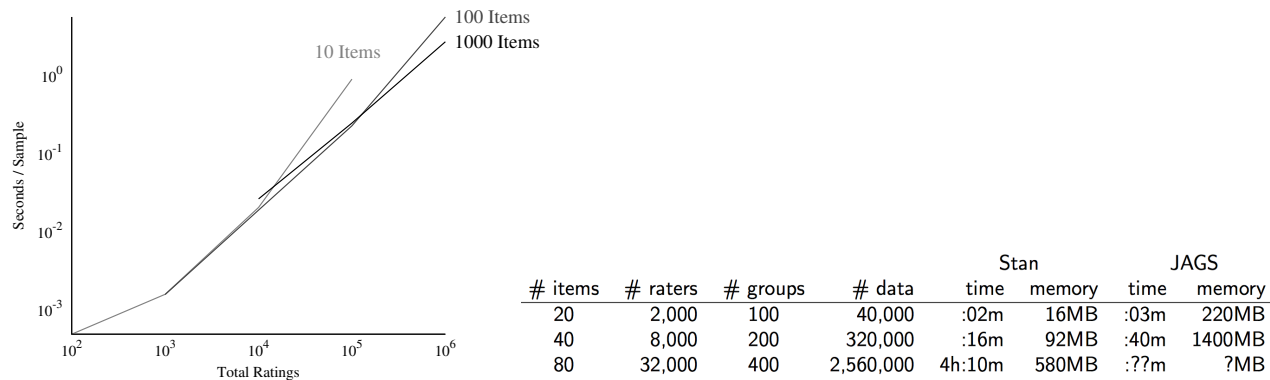


Figure 2: (left) With thousands of items, and millions of total ratings, Stan is able to generate samples from a hierarchical logistic regression model in seconds. With only a few hundred iterations needed for inference, this scaling admits practical implementation of such item-response theory models with large data sets. (right) Stan is twice as fast (and also yields roughly four times as many effective samples) as the previous state of the art, Jags.

unless they can be averaged over as in certain finite mixture models. In addition, Bugs and Jags are based on graphical models, which is a more restricted class than the arbitrary objective functions that can be specified by Stan but which has certain advantages; for example, if a subset of modeled data is missing, a Bugs or Jags model can be re-run as written, whereas the Stan model needs to be altered to consider these missing values as parameters.

Missing data and discrete parameters aside, programming is about as easy in Bugs and Jags as in Stan. In some ways Stan is simpler because its modeling language is imperative (that is, each line of code has a particular effect), whereas Bugs and Jags use declarative languages. On the other hand, there is a learning curve for an existing user of Bugs or Jags to switch to Stan. To ameliorate this problem, the example models in the Jags manual have been translated and are available on the Stan webpage.

Stan is faster for complex models and scales better than Bugs or Jags for large datasets; this is a consequence partly of efficient implementation and use of memory management (Gay, 2005) and partly of Stan’s advanced MCMC algorithm. For example, Figure 2 shows a speed comparison for a hierarchical logistic regression as the problem scales up. Models with matrix parameters (for example, multilevel models with multiple coefficients that vary by group) are particularly slow in Bugs and Jags—Gibbs sampling does not work well with covariance matrices. Stan is not lightning-fast in such settings but, for problems of moderate size, it runs well enough to be practical. For example, the hierarchical time-series model in (Ghitza & Gelman, 2014) required several hours to run in Stan but would not have been fittable at all in other general-purpose Bayesian software.

Bugs and Jags are competitive with Stan on certain problems with conjugate models and low posterior correlation, so that Gibbs sampling is fast and efficient. In any case, none of this is meant as a criticism of Bugs and Jags, which were pathbreaking programs that inspired the development of Stan.

Existing general-purpose optimizers. Stan also contains an optimizer and so it can be compared to various the optimizers that exist in various scientific software packages. We have not yet performed any speed comparisons but we believe Stan should perform well because L-BFGS is known to be efficient, and Stan’s automatic differentiation is fast and is optimized for problems such as normal distributions and logistic models that are common in statistics. We are currently finishing implementing higher-order automatic differentiation in Stan, at which point Stan’s optimizer will

be able to immediately supply Hessians which can be used to compute asymptotic variances.

Preprogrammed approximations to Bayesian inference for specific model classes. Bayesian inference, or approximate Bayesian inference (for example, hierarchical models using marginal posterior modes) have been implemented for various classes of models with applied importance, including:

- Multilevel linear and logistic regressions (lme4 in R (Bates, Maechler, Bolker, & Walker, 2014), gllamm in Stata (Rabe-Hesketh, Skrondal, & Pickles, 2005), HLM (Raudenbush, Bryk, & Congdon, 2004), and Mplus (Muthén & Muthén, 1998–2011));
- Item-response and ideal-point models (mcmcpack in R (Martin, Quinn, & Park, 2011));
- Differential equation models in pharmacology with parameters that vary by person (Nonmem (Beal, Sheiner, Boeckmann, & Bauer, 1989–2009));
- Gaussian process regression (GPstuff in Matlab (Vanhatalo et al., 2013)).

Stan competes with these programs in different ways.

Multilevel regressions, item-response, and ideal-point models can already be fit in Stan, although there remain certain problems of moderate size where approximate methods can give reasonable answers in less time.

Stan can fit hierarchical differential equation models, allowing users more flexibility and a more full accounting for uncertainty, as compared to Nonmem, which uses point estimation for hyperparameters. In such problems, we have found Stan’s full Bayes approach to be more computationally stable, especially in the presence of reasonably informative prior distributions. But there remain many practical examples where Stan remains quite a bit slower than Nonmem, and for these examples it makes sense to have access to both systems.

Gaussian processes are currently more challenging. When the number of elements of the process is large (1000 or more, say), full Bayesian inference for Gaussian processes using Stan can become not just slow but effectively impossible for models such as logistic regressions where there is no analytic solution. In contrast, GPstuff uses an implementation of expectation propagation (Vanhatalo et al., 2013) which is fast enough to work for problems that are too large for Stan to tackle (and much too large for Bugs or Jags).

Programming Bayesian inference directly. A final option, indeed until recently the first option for Bayesian statisticians, is to program one’s model and inference directly in C, R, or Python or using some set of tools such as PyMC (Patil, Huard, & Fonnesbeck, 2010). Such programs have two parts: the specification of the model, and the algorithm used to perform the sampler. Stan makes most such efforts obsolete because it is laborious to hand-write code to compute a log probability density and its gradient, and in many cases the Stan implementation will be faster.

One area where direct coding may still be necessary is when model predictions come from a complex “black box” which cannot easily be translated into Stan. In such cases, it can be easier to build a sampler around the black box rather than to take the code apart and put it inside Stan. There are also certain difficult problems, such as inference for models with unknown normalizing constants, where the form of the model itself must be approximated so that no general purpose algorithm can work.

6. Other features and future developments

Stan has many capacities not discussed in the present article, including a large library of distributions and mathematical functions, random number generation, and the ability for users to write their own functions that can be called in R or by other Stan programs.

The current highest priority of work in Stan is in three directions: *more sophisticated algorithms* such as Riemannian HMC (Betancourt, 2013) for hard problems; *approximate algorithms* including variational inference (Kucukelbir, Ranganath, Gelman, & Blei, 2014), expectation propagation (Gelman et al., 2014), and marginal maximum likelihood for big models fit to big data; and *user-friendly implementations* of workhorse models such as hierarchical regressions and Gaussian processes. Under the hood, important steps in making these algorithms run fast include higher-order automatic differentiation and more efficient implementations of matrix operations. In addition, we are working to make Stan faster, one model at a time, working with colleagues in various applied fields including ecology, pharmacology, astronomy, political science, and education research. For that last project we are developing efficient and user-friendly implementations of linear regression, generalized linear models, hierarchical models, and item-response models.

These implementations offer several advantages beyond what was previously available in R, Stata, and other statistics packages: the Stan implementations should be more stable and scale better for large datasets and large models, include more general model classes (for example, combining hierarchical modeling with ordered multinomial regression, a family that is not, strictly speaking, a generalized linear model but can be implemented easily enough in Stan), and easily allow prior distributions, which can make a big difference for regularization when parameters are only weakly identified. We also would like to extend Stan to allow discrete parameters. The larger goal is to relax the constraints which researchers often face, having to choose among a limited menu of models because of computational restrictions.

References

- Bates, D., Maechler, M., Bolker, B., & Walker, S. (2014). lme4: Linear mixed-effects models using Eigen and S4 [Computer software manual]. Retrieved from <http://CRAN.R-project.org/package=lme4> (R package version 1.1-7)
- Beal, S., Sheiner, L., Boeckmann, A., & Bauer, R. (1989–2009). NONMEM user’s guides [Computer software manual]. Ellicott City, MD.
- Betancourt, M. (2013). Generalizing the no-U-turn sampler to Riemannian manifolds. *arXiv*, 1304.1920. Retrieved from <http://arxiv.org/abs/1304.1920>
- Byrd, R. H., Lu, P., Nocedal, J., & Zhu, C. (1994). A limited memory algorithm for bound constrained optimization. *SIAM Journal on Scientific Computing*, 16, 1190–1208.
- Gay, D. M. (2005). Semiautomatic differentiation for efficient gradient computations. In H. M. Bücker, G. F. Corliss, P. Hovland, U. Naumann, & B. Norris (Eds.), *Automatic differentiation: Applications, theory, and implementations* (Vol. 50, pp. 147–158). New York: Springer. doi: 10.1007/3-540-28438-9_13
- Gelman, A., Carlin, J. B., Stern, H. S., Dunson, D. B., Vehtari, A., & Rubin, D. B. (2013). *Bayesian data analysis* (Third ed.). London: Chapman & Hall/CRC Press.
- Gelman, A., Vehtari, A., Jylänki, P., Robert, C., Chopin, N., & Cunningham, J. P. (2014). Expectation propagation as a way of life. *arXiv*, 1412.4869.
- Ghitza, Y., & Gelman, A. (2014). The great society, Reagan’s revolution, and generations of presidential voting. *To be submitted*.

- Griewank, A., & Walther, A. (2008). *Evaluating derivatives: Principles and techniques of algorithmic differentiation* (2nd ed.). Society for Industrial and Applied Mathematics (SIAM).
- Hoffman, M. D., & Gelman, A. (2014). The no-U-turn sampler: Adaptively setting path lengths in Hamiltonian Monte Carlo. *Journal of Machine Learning Research*, 15, 1593–1623. Retrieved from <http://jmlr.org/papers/v15/hoffman14a.html>
- Kucukelbir, A., Ranganath, R., Gelman, A., & Blei, D. M. (2014). Fully automatic variational inference of differentiable probability models. In *Proceedings of the nips workshop on probabilistic programming*.
- Lewandowski, D., Kurowicka, D., & Joe, H. (2009). Generating random correlation matrices based on vines and extended onion method. *Journal of Multivariate Analysis*, 100, 1989–2001.
- Lunn, D., Spiegelhalter, D., Thomas, A., & Best, N. (2009). The BUGS project: Evolution, critique and future directions. *Statistics in Medicine*, 28(25), 3049–3067.
- Lunn, D., Thomas, A., Best, N., & Spiegelhalter, D. (2000). Winbugs – a Bayesian modelling framework: concepts, structure, and extensibility. *Statistics and Computing*, 10, 325–337.
- Martin, A. D., Quinn, K. M., & Park, J. H. (2011). MCMCpack: Markov chain monte carlo in R. *Journal of Statistical Software*, 42(9), 22. Retrieved from <http://www.jstatsoft.org/v42/i09/>
- Metropolis, N., & Ulam, S. (1949). The Monte Carlo method. *Journal of the American Statistical Association*, 44(247), 335–341.
- Muthén, L., & Muthén, B. (1998–2011). Mplus user’s guide (6th ed.) [Computer software manual]. Los Angeles, CA.
- Neal, R. (2011). MCMC using Hamiltonian dynamics. In S. Brooks, A. Gelman, G. L. Jones, & X.-L. Meng (Eds.), *Handbook of Markov chain Monte Carlo* (pp. 116–162). Chapman and Hall/CRC.
- Patil, A., Huard, D., & Fonnesbeck, C. J. (2010). PyMC: Bayesian stochastic modelling in Python. *Journal of Statistical Software*, 35.
- Plummer, M. (2003). JAGS: A program for analysis of Bayesian graphical models using Gibbs sampling. In *Proceedings of the 3rd international workshop on distributed statistical computing* (Vol. 124, p. 125).
- Rabe-Hesketh, S., Skrondal, A., & Pickles, A. (2005). Maximum likelihood estimation of limited and discrete dependent variable models with nested random effects. *Journal of Econometrics*, 128, 301–323.
- Raudenbush, S., Bryk, A., & Congdon, R. (2004). HLM 6 for Windows [Computer software manual]. Skokie, IL.
- Stan Development Team. (2015). Stan modeling language users guide and reference manual, version 2.6.0 [Computer software manual]. Retrieved from <http://mc-stan.org/>
- Vanhatalo, J., Riihimäki, J., Hartikainen, J., Jylänki, P., Tolvanen, V., & Vehtari, A. (2013). GP-stuff: Bayesian Modeling with Gaussian Processes. *Journal of Machine Learning Research*, 14, 1175–1179.