



Journal of Statistical Software

MMMMMM YYYY, Volume VV, Issue II.

<http://www.jstatsoft.org/>

Stan: A Probabilistic Programming Language

Bob Carpenter
Columbia University

Andrew Gelman
Columbia University

Matt Hoffman
Adobe Research Labs

Daniel Lee
Columbia University

Ben Goodrich
Columbia University

Michael Betancourt
University College London

Marcus A. Brubaker
TTI-Chicago

Jiqiang Guo
Columbia University

Peter Li
Columbia University

Allen Riddell
Dartmouth College

Abstract

Stan is a probabilistic programming language for specifying statistical models. A Stan program imperatively defines a log probability function over parameters conditioned on specified data and constants. As of version 2.2.0, Stan provides full Bayesian inference for continuous-variable models through Markov chain Monte Carlo methods such as the No-U-Turn sampler, an adaptive form of Hamiltonian Monte Carlo sampling. Penalized maximum likelihood estimates are calculated using optimization methods such as the Broyden-Fletcher-Goldfarb-Shanno algorithm.

Stan is also a platform for computing log densities and their gradients and Hessians, which can be used in alternative algorithms such as variational Bayes, expectation propagation, and marginal inference using approximate integration. To this end, Stan is set up so that the densities, gradients, and Hessians, along with intermediate quantities of the algorithm such as acceptance probabilities, are easily accessible.

Stan can be called from the command line, through R using the **RStan** package, or through Python using the **PyStan** package. All three interfaces support sampling or optimization-based inference and analysis, and **RStan** and **PyStan** also provide access to log probabilities, gradients, Hessians, and data I/O.

Keywords: probabilistic program, Bayesian inference, algorithmic differentiation, Stan.

1. Introduction

The goal of the Stan project is to provide a flexible probabilistic programming language for statistical modeling along with a suite of inference tools for fitting models that are robust, scalable, and efficient.

Stan differs from BUGS (Lunn, Thomas, and Spiegelhalter 2000; Lunn, Spiegelhalter, Thomas, and Best 2009; Lunn, Jackson, Best, Thomas, and Spiegelhalter 2012) and JAGS (Plummer 2003) in two primary ways. First, Stan is based on a new imperative probabilistic programming language that is more flexible and expressive than the declarative graphical modeling languages underlying BUGS or JAGS, in ways **such as declaring variables with types and supporting local variables and conditional statements**. Second, Stan’s Markov chain Monte Carlo (MCMC) techniques are based on Hamiltonian Monte Carlo (HMC), a more efficient and robust sampler than Gibbs sampling or Metropolis-Hastings for models with complex posteriors.¹

Stan has interfaces for the command-line shell (**CmdStan**), Python (**PyStan**), and R (**RStan**), and runs on Windows, Mac OS X, and Linux, and is open-source licensed.

The next section provides an overview of how Stan works by way of an extended example, after which the details of Stan’s programming language and inference mechanisms are provided.

2. Core Functionality

This section describes the use of Stan from the command line for estimating a Bayesian model using both MCMC sampling for full Bayesian inference and optimization to provide a point estimate at the posterior mode.

2.1. Model for estimating a Bernoulli parameter

Consider estimating the chance of success parameter for a Bernoulli distribution based on a sequence of observed binary outcomes. Figure 1 provides an implementation of such a model in Stan.² The model treats the observed binary data, $y[1], \dots, y[N]$, as independent and identically distributed, with success probability `theta`. The vectorized likelihood statement can also be coded using a loop as in BUGS, although it will run more slowly than the vectorized form:

¹Neal (2011) analyzes the scaling benefit of HMC with dimensionality. Hoffman and Gelman (2014) provide practical comparisons of Stan’s adaptive HMC algorithm with Gibbs, Metropolis, and standard HMC samplers.

²This model is available in the Stan source distribution in `src/models/basic_estimators/bernoulli.stan`.

```

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

```

Figure 1: *Model for estimating a Bernoulli parameter.*

```

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

```

A `beta(1,1)` (i.e., uniform) prior is placed on `theta`, although there is no special behavior for conjugate priors in `Stan`. The prior could be dropped from the model altogether because parameters start with uniform distributions on their support, here constrained to be between 0 and 1 in the parameter declaration for `theta`.

2.2. Data format

Data for running `Stan` from the command line can be included in R dump format. All of the variables declared in the data block of the `Stan` program must be defined in the data file. For example, 10 observations for the model in Figure 1 could be encoded as³

³This data file is provided with the `Stan` distribution in file `src/models/basic_estimators/bernoulli.R.stan`.

```
N <- 10
y <- c(0,1,0,0,0,0,0,0,0,1)
```

This defines the contents of two variables, an integer `N` and a 10-element integer array `y`. The variable `N` is declared in the data block of the program as being an integer greater than or equal to zero; the variable `y` is declared as an integer array of size `N` with entries between 0 and 1 inclusive.

In **RStan** and **PyStan**, data can also be passed directly through memory without the need to read or write to a file.

2.3. Compiling the model

After a C++ compiler and `make` are installed,⁴ the Bernoulli model in Figure 1 can be translated to C++ and compiled with a single command. First, the directory must be changed to `$stan`, which we use as a shorthand for the directory in which **Stan** was unpacked.⁵

```
> cd $stan
> make src/models/basic_estimators/bernoulli
```

This produces an executable file `bernoulli` (`bernoulli.exe` on Windows) on the same path as the model. Forward slashes can be used with `make` on Windows.

2.4. Running the sampler

Command to sample from the model

The executable can be run with default options by specifying a path to the data file. The first command in the following example changes the current directory to that containing the model, which is where the data resides and where the executable is built. From there, the path to the data is just the file name `bernoulli.data.R`.

```
> cd $stan/src/models/basic_estimators
> ./bernoulli sample data file=bernoulli.data.R
```

For Windows, the `./` before the command should be removed. This call specifies that sampling should be performed with the model instantiated using the data in the specified file.

Terminal output from sampler

The output is as follows, starting with a summary of the command-line options used, including defaults; these are also written into the samples file as comments.

⁴Appropriate versions are built into Linux. The **RTools** package suffices for Windows; it is available from <http://cran.r-project.org/bin/windows/Rtools/>. The **Xcode** package contains everything needed for the Mac; see <https://developer.apple.com/xcode/> for more information.

⁵Before the first model is built, `make` must build the model translator (target `bin/stanc`) and posterior summary tool (target `bin/print`), along with an optimized version of the C++ library (target `bin/libstan.a`). Please be patient and consider `make` option `-j2` or `-j4` (or higher) to run in the specified number of processes if two or four (or more) computational cores are available.

```

method = sample (Default)
  sample
    num_samples = 1000 (Default)
    num_warmup = 1000 (Default)
    save_warmup = 0 (Default)
    thin = 1 (Default)
  adapt
    engaged = 1 (Default)
    gamma = 0.050000000000000003 (Default)
    delta = 0.80000000000000004 (Default)
    kappa = 0.75 (Default)
    t0 = 10 (Default)
    init_buffer = 75 (Default)
    term_buffer = 50 (Default)
    window = 25 (Default)
  algorithm = hmc (Default)
    hmc
      engine = nuts (Default)
      nuts
        max_depth = 10 (Default)
        metric = diag_e (Default)
        stepsize = 1 (Default)
        stepsize_jitter = 0 (Default)
  id = 0 (Default)
  data
    file = bernoulli.data.R
  init = 2 (Default)
  random
    seed = 4294967295 (Default)
  output
    file = output.csv (Default)
    diagnostic_file = (Default)
    refresh = 100 (Default)

```

Gradient evaluation took 4e-06 seconds
 1000 transitions using 10 leapfrog steps per transition would take
 0.04 seconds.

Adjust your expectations accordingly!

```

Iteration:    1 / 2000 [ 0%] (Warmup)
Iteration:  100 / 2000 [ 5%] (Warmup)
...
Iteration: 1000 / 2000 [ 50%] (Warmup)
Iteration: 1001 / 2000 [ 50%] (Sampling)
...
Iteration: 2000 / 2000 [100%] (Sampling)

```

```

Elapsed Time: 0.00932 seconds (Warm-up)
              0.016889 seconds (Sampling)
              0.026209 seconds (Total)

```

The sampler configuration parameters are echoed, here they are all default values other than the data file.

The command-line parameters marked `Default` may be explicitly set on the command line. Each value is preceded by the full path to it in the hierarchy; for instance, to set the maximum depth for the no-U-turn sampler, the command would be the following, where backslash indicates a continued line.

```

> ./bernoulli sample \
  algorithm=hmc engine=nuts max_depth=5 \
  data max_depthfile=bernoulli.data.R

```

Help

A description of all configuration parameters including default values and constraints is available by executing

```

> ./bernoulli help-all

```

The sampler and its configuration are described at greater length in the manual ([Stan Development Team 2014](#)).

Samples file output

The output CSV file, written by default to `output.csv`, starts with a summary of the configuration parameters for the run.

```

# stan_version_major = 2
# stan_version_minor = 1
# stan_version_patch = 0
# model = bernoulli_model
# method = sample (Default)
#   sample
#     num_samples = 1000 (Default)
#     num_warmup = 1000 (Default)
#     save_warmup = 0 (Default)
#     thin = 1 (Default)
#   adapt
#     engaged = 1 (Default)
#     gamma = 0.05000000000000003 (Default)
#     delta = 0.8000000000000004 (Default)
#     kappa = 0.75 (Default)
#     t0 = 10 (Default)
#     init_buffer = 75 (Default)
#     term_buffer = 50 (Default)

```

```

#       window = 25 (Default)
#   algorithm = hmc (Default)
#       hmc
#       engine = nuts (Default)
#       nuts
#       max_depth = 10 (Default)
#       metric = diag_e (Default)
#       stepsize = 1 (Default)
#       stepsize_jitter = 0 (Default)
# id = 0 (Default)
# data
#   file = bernoulli.data.R
# init = 2 (Default)
# random
#   seed = 847896134
# output
#   file = output.csv (Default)
#   diagnostic_file = (Default)
#   refresh = 100 (Default)

```

Stan's behavior is fully specified by these configuration parameters, almost all of which have default values. The `sample` configuration

By using the same version of `Stan` and these configuration parameters, exactly the same output file can be reproduced. The pseudorandom numbers generated by the sampler are fully determined by the seed (here randomly generated based on the time of the run, with value 847896134) and the identifier (here 0). The identifier is used to advance the underlying pseudorandom number generator a sufficient number of values that using multiple chains with the same seed and different identifiers will draw from different subsequences of the pseudorandom number stream determined by the seed.

The output continues with a CSV header naming the columns of the output. For the default NUTS sampler in `Stan 2.2.0`, these are

```
lp__,accept_stat__,stepsize__,treedepth__,n_divergent__,theta
```

The values headed by `lp__` are the log densities (up to an additive constant), `accept_stat__` are the Metropolis acceptance probabilities averaged over samples in the slice used by the no-U-turn sampler, `stepsize__` is the leapfrog integrator's step size for simulating the Hamiltonian, `treedepth__` is the depth of tree explored by the no-U-turn sampler, and `n_divergent__` is the number of iterations leading to a numerical instability during integration (e.g., numerical overflow or a positive-definiteness violation).

for each iteration.⁶ The column `stepsize__` indicates the step size (i.e., time interval) of the simulated trajectory, while the column `treedepth__` gives the tree depth for NUTS, defined

⁶Acceptance is the usual notion for a Metropolis sampler such as HMC ([Metropolis, Rosenbluth, Rosenbluth, Teller, and Teller 1953](#)). For NUTS, the acceptance statistic is defined as the average acceptance probabilities of all possible samples in the proposed tree; NUTS itself uses a slice sampling algorithm for rejection ([Neal 2003](#); [Hoffman and Gelman 2014](#)).

as the log base 2 of the total number of steps in the trajectory. The rest of the header will be the names of parameters; in this example, `theta` is the only parameter.

Next, the results of adaptation are printed as comments.

```
# Adaptation terminated
# Step size = 0.783667
# Diagonal elements of inverse mass matrix:
# 0.517727
```

By default, Stan uses the NUTS sampler with a diagonal mass matrix. The mass matrix is estimated, roughly speaking, by regularizing the sample covariance of the latter half of the warmup samples; see (Stan Development Team 2014) for full details. A dense mass matrix may also be estimated, or the mass matrix may be set to the unit matrix.

The rest of the file contains samples, one per line, matching the header; here the parameter `theta` is the final value printed on each line, and each line corresponds to a sample. The warmup samples are not included by default, but may be included with the appropriate command-line invocation of the executable. The file ends with comments reporting the elapsed time.

```
-7.19297,1,0.783667,1,0,0.145989
-8.2236,0.927238,0.783667,1,0,0.0838792
...
-7.48489,0.738509,0.783667,0,0,0.121812
-7.40361,0.995299,0.783667,1,0,0.407478
-9.49745,0.771026,0.783667,2,0,0.0490488
-9.11119,1,0.783667,0,0,0.0572588
-7.20021,0.979883,0.783667,1,0,0.14527

# Elapsed Time: 0.010849 seconds (Warm-up)
#               0.01873 seconds (Sampling)
#               0.029579 seconds (Total)
```

It is evident from the values sampled for `theta` in the last column that there is a high degree of posterior uncertainty in the estimate of `theta` from the ten data points in the data file.

The log probabilities reported in the first column include not only the model log probabilities but also the Jacobian adjustment resulting from the transformation of the variables to unconstrained space. Here, that is the absolute derivative of the inverse logistic function; see (Stan Development Team 2014) for full details on all of the transforms and their Jacobians.

2.5. Sampler output analysis

Before performing output analysis, we recommend generating multiple independent chains in order to more effectively monitor convergence; see (Gelman and Rubin 1992) for more analysis. Three more chains of samples can be created as follows.

```
./bernoulli sample data file=bernoulli.data.R random seed=847896134 \
            id=1 output file=output1.csv
```


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 (0.0108, 0.0130, 0.0110, 0.0110) seconds, 0.0459 seconds total

Sampling took (0.0187, 0.0190, 0.0168, 0.0176) seconds, 0.0722 seconds total

	Mean	MCSE	StdDev	5%	50%	95%
lp__	-7.28	1.98e-02	0.742	-8.85e+00	-6.99	-6.75
accept_stat__	0.909	4.98e-03	0.148	5.70e-01	0.971	1.00
stepsize__	0.927	7.45e-02	0.105	7.84e-01	1.00	1.05
treedepth__	0.437	1.03e-02	0.551	0.00e+00	0.000	1.00
n_divergent__	0.000	0.00e+00	0.000	0.00e+00	0.000	0.000
theta	0.254	3.25e-03	0.122	7.58e-02	0.238	0.479

	N_Eff	N_Eff/s	R_hat
lp__	1404	19447	1.00e+00
accept_stat__	887	12297	1.02e+00
stepsize__	2.00	27.7	5.56e+13
treedepth__	2856	39572	1.01e+00
n_divergent__	4000	55424	nan
theta	1399	19382	1.00e+00

Figure 2: Output of bin/print for the Bernoulli estimation model in Figure 1.

```
./bernoulli sample data file=bernoulli.data.R random seed=847896134 \  
id=2 output file=output2.csv  
./bernoulli sample data file=bernoulli.data.R random seed=847896134 \  
id=3 output file=output3.csv
```

These calls illustrate how additional parameters are specified directly on the command line following the hierarchy given in the output. The backslash (\) at the end of each line indicates that the command continues on the last line; a caret (^) should be used in Windows.

The chains can be safely run in parallel under different processes; details of parallel execution depend on the operating system and the shell or terminal program. Note that, although the same seed is used for each chain, the random numbers will in fact be independent as the chain identifier is used to skip the pseudorandom number generator ahead.

Stan supplies a command-line program bin/print to summarize the output of one or more MCMC chains. Given a directory containing output from sampling,

```
> ls output*.csv
```

```
output.csv          output1.csv          output2.csv          output3.csv
```

posterior summaries are printed using

```
> $stan/bin/print output*.csv
```

The output is shown in Figure 2.⁷ Each row of the output summarizes a different value whose name is provided in the first column. These correspond to the columns in the output CSV files. The analysis includes estimates of the posterior mean (**Mean**) and standard deviation (**StdDev**). The median (50%) and 90% posterior interval (5%, 95%) are also displayed.

The remaining columns in the output provide an analysis of the sampling and its efficiency. The convergence diagnostic that is built into the **bin/print** command is the estimated potential scale reduction statistic \hat{R} (**Rhat**); its value should be close to 1.0 when the chains have all converged to the same stationary distribution. Stan uses a more conservative version of \hat{R} than is usual in packages such as **Coda** (Plummer, Best, Cowles, and Vines 2006), first splitting each chain in half to diagnose nonstationary chains; see (Gelman, Carlin, Stern, Dunson, Vehtari, and Rubin 2013) and (Stan Development Team 2014) for detailed definitions.

The column **N_eff** is the number of effective samples in a chain. Because MCMC methods produce correlated samples in each chain, estimates such as posterior means are not as accurate as they would be with truly independent samples. The number of effective samples is an estimate of the number of independent samples that would lead to the same accuracy. The Monte Carlo standard error (MCSE) is an estimate of the error in estimating the posterior mean based on dividing the posterior standard deviation estimate by the square root of the number of effective samples (**sd** / **sqrt(n_eff)**). Geyer (2011) provides a thorough introduction to effective sample size and MCSE estimation. Stan uses the more conservative estimates based on both within-chain and cross-chain convergence; see (Gelman *et al.* 2013) and (Stan Development Team 2014) for motivation and definitions.

Because estimation accuracy is governed by the square root of the number of effective samples, effective samples per second (or seconds per effective sample) is the most relevant statistic for comparing the efficiency of sampler implementations. Compared to BUGS and JAGS, Stan is often relatively slow per iteration but relatively fast per effective sample.

In this example, the estimated number of effective samples per parameter (**n_eff**) is 1399, which far more than we typically need for inference. The posterior mean here is estimated to be 0.254 with an MCSE of 0.00325. Because the model is conjugate, the exact posterior is known to be $p(\theta|y) = \text{Beta}(3, 9)$. Thus the posterior mean of θ is $3/(3 + 9) = 0.25$ and the posterior mode of θ is $(3 - 1)/(3 + 9 - 2) = 0.2$.

2.6. Posterior mode estimates

Posterior modes with optimization

The posterior mode of a model can be found by using one of Stan's built-in optimizers. The following command invokes optimization for the Bernoulli model using all default configuration parameters.

```
> ./bernoulli optimize data file=bernoulli.data.R
```

⁷Aligning columns when printing rows of varying scales presents a challenge. For each column, the program calculates the maximum number of digits required to print an entry in that column with the specified precision. For example, a precision of 2 for the number -0.000012 requires nine characters (-0.000012) to print without scientific notation versus seven digits with (-1.2e-5). If the discrepancy is above a fixed threshold, scientific notation is used. Compare the results in the **mean** column versus the **sd** column.

```

method = optimize
  optimize
    algorithm = bfgs (Default)
      bfgs
        init_alpha = 0.001 (Default)
        tol_obj = 1e-08 (Default)
        tol_grad = 1e-08 (Default)
        tol_param = 1e-08 (Default)
      iter = 2000 (Default)
      save_iterations = 0 (Default)
id = 0 (Default)
data
  file = bernoulli.data.R
init = 2 (Default)
random
  seed = 4294967295 (Default)
output
  file = output.csv (Default)
  diagnostic_file = (Default)
  refresh = 100 (Default)

initial log joint probability = -12.4873
  Iter      log prob      ||dx||      ||grad||      alpha  # evals  Notes
    7      -5.00402    8.61455e-07    1.25715e-10        1        10
Optimization terminated normally:
  Convergence detected: change in objective function was below
  tolerance

```

The final lines of the output indicate normal termination after seven iterations by convergence of the objective function (here the log probability) to the default tolerance of $1e-08$. The final log probability (`log prob`), length of the difference between the current iteration's value of the parameter vector and the previous value (`||dx||`), and the length of the gradient vector (`||grad||`).

The optimizer terminates when any of the log probability, gradient, or parameter values are within their specified tolerance. The default optimizer uses the Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm, a quasi-Newton method which employs exactly computed gradients and an efficient approximation to the Hessian; see (Nocedal and Wright 2006) for a textbook exposition of the BFGS algorithm.

Optimizer output file

By default, optimizations results are written into `output.csv`, which is a valid CSV file.

```

# stan_version_major = 2
# stan_version_minor = 1
# stan_version_patch = 0
# model = bernoulli_model

```

```

# method = optimize
#   optimize
#     algorithm = bfgs (Default)
#       bfgs
#         init_alpha = 0.001 (Default)
#         tol_obj = 1e-08 (Default)
#         tol_grad = 1e-08 (Default)
#         tol_param = 1e-08 (Default)
#       iter = 2000 (Default)
#       save_iterations = 0 (Default)
# id = 0 (Default)
# data
#   file = bernoulli.data.R
# init = 2 (Default)
# random
#   seed = 777510854
# output
#   file = output.csv (Default)
#   diagnostic_file = (Default)
#   refresh = 100 (Default)
lp__,theta
-5.00402,0.2000000000125715

```

As with the sampler output, the configuration of the optimizer is dumped as CSV comments (lines beginning with #). Then there is a header, listing the log probability, `lp__`, and the single parameter name, `theta`. The next line shows that the posterior mode for `theta` is 0.2000000000125715, matching the true posterior mode of 0.20 very closely.

Optimization is carried out on the unconstrained parameter space, but without the Jacobian adjustment to the log probability. This ensures modes are defined with respect to the constrained parameter space as declared in the parameters block and used in the model specification. The need to suppress the Jacobian to match the scaling of the declared parameters highlights the sensitivity of posterior modes to parameter transforms.

2.7. Diagnostic mode

Stan provides a diagnostic mode that evaluates the log probability and gradient calculations at the initial parameter values (either user supplied or generated randomly based on the specified or default seed).

```

> ./bernoulli diagnose data file=bernoulli.data.R

method = diagnose
diagnose
  test = gradient (Default)
  gradient
    epsilon = 9.999999999999995e-07 (Default)
    error = 9.999999999999995e-07 (Default)

```

```

id = 0 (Default)
data
  file = bernoulli.data.R
init = 2 (Default)
random
  seed = 4294967295 (Default)
output
  file = output.csv (Default)
  diagnostic_file = (Default)
  refresh = 100 (Default)

```

TEST GRADIENT MODE

Log probability=-6.74818

param idx	value	model	finite diff	error
0	-1.1103	0.0262302	0.0262302	-3.81445e-10

Here, a random initialization is used and the initial log probability is -6.74818 and the single parameter `theta`, here represented by index 0, has a value of -1.1103 on the unconstrained scale. The derivative supplied by the model and by a finite differences calculation are the same to within $-3.81445\text{e-}10$. Non-finite log probability values or derivatives indicate a problem with the model in terms of constraints on parameter values or function inputs being violated, boundary conditions in functions, and sometimes overflow or underflow issues with floating-point calculations. Errors between the model's gradient calculation and finite differences can indicate a bug in Stan's algorithmic differentiation for a function in the model.

2.8. Roadmap for the Rest of the Paper

Now that the key functionality of Stan has been demonstrated, the remaining sections cover specific aspects of Stan's architecture. Section 3 covers variable data type declarations as well as expressions and type inference, Section 4 describes the top-level blocks and execution of a Stan program, Section 5 lays out the available statements, and Section 6 the built-in math, matrix, and probability function library. Section 7 lays out MCMC and optimization-based inference. There are two appendices, Appendix A outlining the development process and Appendix B detailing the library dependencies.

3. Data types

All expressions in Stan are statically typed, including variables. This means their type is declared at compile time as part of the model, and does not change throughout the execution of the program. This is the same behavior as is found in compiled programming languages such as C(++), Fortran, and Java, but is unlike the behavior of interpreted languages such as BUGS, R, and Python. Statically typing the variables (as well as declaring them in appropriate blocks based on usage) makes Stan programs easier to read and easier to debug by making explicit the modeling decisions and expression types.

3.1. Primitive types

The primitive types of Stan are `real` and `int`, which are used to represent continuous and integer values. These values are represented directly in C++ as types `double` and `int`. Integer expressions can be used anywhere a real value is required, but not *vice-versa*.

3.2. Vector and matrix types

Stan supports vectors, row vectors, and matrices with the usual access operations. Indexing for vector, matrix, and array types starts from one.

Vectors are declared with their sizes and matrices with their number of rows and columns. Vector, row vector, and matrix elements are accessed using bracket notation, as in `y[3]` for the third element of a vector or row vector and `a[2,3]` for the element in the third column of the second row of a matrix. Indexing begins from 1. The notation `a[2]` accesses the second row of matrix `a`.

All vector and matrix types contain real values and may not be declared to contain integers. Collections of integers are represented using arrays.

3.3. Array types

An array may have entries of any other type. For example, arrays of integers and reals are allowed, as are arrays of vectors or arrays of matrices.

Higher-dimensional arrays are intrinsically arrays of arrays. An entry in a two-dimensional array `y` may be accessed as `y[1,2]`. The expression `y[1]` by itself denotes the one-dimensional array whose values correspond to the first row of `y`. Thus `y[1][2]` has the same value as `y[1,2]`.⁸

Unlike integers, which may be used where real values are required, arrays of integers may not be used where real arrays are required.⁹

The manual contains a chapter discussing the efficiency tradeoffs and motivations for separating arrays and matrices.

3.4. Constrained variable types

Variables may be declared with constraints. The constraints have different effects depending on the block in which the variable is declared.

Integer and real types may be provided with lower bounds, upper bounds, or both. This includes the types used in arrays, and the real types used in vectors and matrices.

Vector types may be constrained to be unit simplexes (all entries non-negative and summing to 1), unit length vectors (sum of squares is 1), or ordered (entries are in ascending order), positive ordered (entries in ascending order, all non-negative), using the types `simplex[K]`, `unit_vector[K]`, `ordered[K]`, or `positive_ordered[K]`, where `K` is the size of the vector.

Matrices may be constrained to be covariance matrices (symmetric, positive definite) or correlation matrices (symmetric, positive definite, unit diagonal), using the types `cov_matrix[K]`

⁸Arrays are stored in row-major order and matrices in column-major order.

⁹In the language of type theory, Stan arrays are not covariant. This follows the behavior of both arrays and standard library containers in C++.

and `corr_matrix[K]`.

3.5. Expressions

The syntax of **Stan** is defined in terms of expressions and statements. Expressions denote values of a particular type. Statements represent operations such as assignment and sampling as well as control structures such as for loops and conditionals.

Stan provides the usual kinds of expressions found in programming languages. This includes variables, literals denoting integers, real values or strings, binary and unary operators over expressions, and function application.

Type inference

The type of a numeric literal is determined by whether or not it contains a period or scientific notation; for example, 20 has type `int` whereas 20.0 and 2e+1 have type `real`.

The type of applying an operator or a function to one or more expressions is determined by the available signatures for the function. For example, the multiplication operator (`*`) has a signature that maps two `int` arguments to an `int` and two `real` arguments to a `real` result. Another signature for the same operator maps a `row_vector` and a `vector` to a `real` result.

Type promotion

If necessary, an integer type will be promoted to a `real` value. For example, multiplying an `int` by a `real` produces a `real` result by promoting the `int` argument to a `real`.

4. Top-Level Blocks and Program Execution

In the rest of this paper, we will concentrate on the modeling language and how compiled models are executed. These details are the same whether a **Stan** model is being used by one of the built-in samplers or optimizers or being used externally by a user-defined sampler or optimizer.

We begin with an example that will be used throughout the rest of this section. (Gelman *et al.* 2013, Section 5.1) define a hierarchical model of the incidence of tumors in rats in control groups across trials; a very similar model is defined for mortality rates in pediatric surgeries across hospitals in (Lunn *et al.* 2000, 2009, Examples, Volume 1). A **Stan** implementation is provided in Figure 3. In the rest of this section, we will walk through what the meaning of the various blocks are for the execution of the model.

4.1. Data block

A **Stan** program starts with an (optional) data block, which declares the data required to fit the model. This is a very different approach to modeling and declarations than in **BUGS** and **JAGS**, which determine which variables are data and which are parameters at run time based on the shape of the data input to them. These declarations make it possible to compile **Stan** to much more efficient code.¹⁰ Missing data models may still be coded in **Stan**, but

¹⁰The speedup is because coding data variables as `double` types in C++ is much faster than promoting all values to algorithmic differentiation class variables.

```

data {
  int<lower=0> J;                // number of items
  int<lower=0> y[J];             // number of successes for j
  int<lower=0> n[J];             // number of trials for j
}
parameters {
  real<lower=0,upper=1> theta[J]; // chance of success for j
  real<lower=0,upper=1> lambda;    // prior mean chance of success
  real<lower=0.1> kappa;           // prior count
}
transformed parameters {
  real<lower=0> alpha;             // prior success count
  real<lower=0> beta;              // prior failure count
  alpha <- lambda * kappa;
  beta <- (1 - lambda) * kappa;
}
model {
  lambda ~ uniform(0,1);           // hyperprior
  kappa ~ pareto(0.1,1.5);         // hyperprior
  theta ~ beta(alpha,beta);        // prior
  y ~ binomial(n,theta);           // likelihood
}
generated quantities {
  real<lower=0,upper=1> avg;        // avg success
  int<lower=0,upper=1> above_avg[J]; // true if j is above avg
  int<lower=1,upper=J> rnk[J];      // rank of j
  int<lower=0,upper=1> highest[J];  // true if j is highest rank
  avg <- mean(theta);
  for (j in 1:J)
    above_avg[j] <- (theta[j] > avg);
  for (j in 1:J) {
    rnk[j] <- rank(theta,j) + 1;
    highest[j] <- rnk[j] == 1;
  }
}

```

Figure 3: *Hierarchical binomial model with posterior inferences, coded in Stan.*

the missing values must be declared as parameters; see (Stan Development Team 2014) for examples of missing data, censored data, and truncated data models.

In the model in Figure 3, the data block declares an integer variable `J` for the number of groups in the hierarchical model. The arrays `y` and `n` have size `J`, with `y[j]` being the number of positive outcomes in `n[j]` trials.

All of these variables are declared with a lower-bound constraint restricting their values to be greater than or equal to zero. Stan's constraint language is not strong enough to restrict each `y[j]` to be less than or equal to `n[j]`.

The data for a Stan model is read in once as the C++ object representing the model is constructed. After the data is read in, the constraints are validated. If the data does not satisfy the declared constraints, the model will throw an exception with an informative error message, which is displayed to the user in the command-line, R, and Python interfaces.

4.2. Transformed data block

The model in Figure 3 does not have a transformed data block. A transformed data block may be used to define new variables that can be computed based on the data. For example, standardized versions of data can be defined in a transformed data block or Bernoulli trials can be summed to model as binomial. Any constant data can also be defined in the transformed data block.

The transformed data block starts with a sequence of variable declarations and continues with a sequence of statements defining the variables. For example, the following transformed data block declares a vector `x_std`, then defines it to be the standardization of `x`:

```
transformed data {
  vector[N] x_std;
  x_std <- (x - mean(x)) / sd(x);
}
```

The transformed data block is executed during construction, after the data is read in. Any data variables declared in the data block may be used in the variable declarations or statements. Transformed data variables may be used after they are declared, although care must be taken to ensure they are defined before they are used. Any constraints declared on transformed data variables are validated after all of the statements are executed, with execution terminating with an informative error message at the first variable with an invalid value.

4.3. Parameter block

The parameter block in the program in Figure 3 defines three parameters. The parameter `theta[j]` represents the probability of success in group `j`. The prior on each `theta[j]` is parameterized by a prior mean chance of success `lambda` and prior count `kappa`. Both `theta[j]` and `lambda` are constrained to fall between zero and one, whereas `kappa` is constrained to be greater than or equal to 0.1 to match the support of the Pareto hyperprior it receives in the model block.

The parameter block is executed every time the log probability is evaluated. This may be multiple times per iteration of a sampling or optimization algorithm.

Implicit change of variables to unconstrained space

The probability distribution defined by a **Stan program is intended to have unconstrained support (i.e., no points of zero probability)**, which greatly simplifies the task of writing samplers or optimizers. To achieve unbounded support, variables declared with constrained support are transformed to an unconstrained space. For instance, variables declared on $[0, 1]$ are log-odds transformed and non-negative variables declared to fall in $[0, \infty)$ are log transformed. More complex transforms are required for simplexes (a reverse stick-breaking transform) and covariance and correlation matrices (Cholesky factorization). The dimensionality of the resulting probability function may change as a result of the transform. For example, a $K \times K$ covariance matrix requires only $\binom{K}{2} + K$ unconstrained parameters, and a K -simplex requires only $K - 1$ unconstrained parameters.

The unconstrained parameters over which the model is defined are inverse transformed back to their constrained forms before executing the model code. To account for the change of variables, the log absolute Jacobian determinant of the inverse transform is added to the overall log probability function.¹¹ The gradients of the log probability function exposed include the Jacobian term.

There is no validation required for the parameter block because the variable transforms are guaranteed to produce values that satisfy the declared constraints.

4.4. Transformed parameters block

The transformed parameters block allows users to define transforms of parameters within a model. Following the model in (Gelman *et al.* 2013), the example in Figure 3 uses the transformed parameter block to define transformed parameters **alpha** and **beta** for the prior success and failure counts to use in the beta prior for **theta**.

Following the same convention as the transformed data block, the (optional) transformed parameter block begins with declarations of the transformed parameters, followed by a sequence of statements defining them. Variables from previous blocks as well as the transformed parameters block may be used. In the example, the prior success and failure counts **alpha** and **beta** are defined in terms of the prior mean **lambda** and total prior count **kappa**.

The transformed parameter block is executed after the parameter block. Constraints are validated after all of the statements defining the transformed parameters have executed. Failure to validate a constraint results in an exception being thrown, which halts the execution of the log probability function. The log probability function can be defined to return negative infinity or the special not-a-number value, both of which are available through built-in functions and may be passed to the `increment_log_prob` function (see below).

If transformed parameters are used on the left-hand side of a sampling statement, it is up to the user to add the appropriate log absolute Jacobian determinant adjustment to the log probability accumulator. For instance, a lognormal variate could be generated as follows without the built-in `lognormal` density function using the normal density as

```
parameters {
  real<lower=0> u;
```

¹¹For optimization, the Jacobian adjustment is suppressed to guarantee the optimizer finds the maximum of the log probability function on the constrained parameters. The calculation of the Jacobian is controlled by a template parameter in the C++ code generated for a model.

Need to
understand
this better

```

...
transformed parameters {
  real v;
  v <- log(u);
  // log absolute Jacobian determinant adjustment
  increment_log_prob(u);
}
model {
  v ~ normal(0,1);
}

```

The transform is $f(u) = \log u$, the inverse transform is $f^{-1}(v) = \exp v$, so the absolute log Jacobian determinant is $|\frac{d}{dv} \exp v| = \exp v = u$. Whenever a transformed parameter is used on the left side of a sampling statement, a warning is printed to remind the user of the need for a Jacobian adjustment for the change of variables.

The `increment_log_prob` statement is used to add a term to the total log probability function defined by the model block and the log absolute Jacobian determinants of the transforms. The variable `lp_`, representing the currently accumulated total log density, may not be assigned to directly.

Values of transformed parameters are saved in the output along with the parameters. As an alternative, local variables can be used to define temporary values that do not need to be saved.

4.5. Model block

The purpose of the model block is to define the log probability function on the constrained parameter space. The example in Figure 3 has a simple model containing four sampling statements. The hyperprior on the prior mean `lambda` is uniform, and the hyperprior on the prior count `kappa` is a Pareto distribution with lower-bound of support at 0.1 and shape 1.5, leading to a probability of $\kappa > 0.1$ proportional to $\kappa^{-5/2}$. Note that the hierarchical prior on `theta` is vectorized: each element of `theta` is drawn independently from a beta distribution with prior success count `alpha` and prior failure count `beta`. Both `alpha` and `beta` are transformed parameters, but because they are only used on the right-hand side of a sampling statement do not require a Jacobian adjustment of their own. The likelihood function is also vectorized, with the effect that each success count `y[i]` is drawn from a binomial distribution with number of trials `n[i]` and chance of success `theta[i]`. In vectorized sampling statements, single values may be repeated as many times as necessary.

The model block is executed after the transformed parameters block every time the log probability function is evaluated.

Implicit uniform priors

The default distribution for a variable is uniform over its declared (constrained) support. For instance, a variable declared with a lower bound of 0 and an upper bound of 1 implicitly receives a `Uniform(0,1)` distribution. These implicit uniform priors are improper if the variable has unbounded support. For instance, the uniform distributions over real values with upper and lower bounds, simplexes and correlation matrices is proper, but the uniform distribution

over unconstrained or one-side constrained reals, ordered vectors or covariance matrices are not proper.

Stan does not require proper priors, but if the posterior is improper, Stan will halt with an error message.¹²

4.6. Generated quantities block

The (optional) generated quantities allows values that depend on parameters and data, but do not affect estimation, to be defined efficiently. The generated quantities block is called only once per sample, not once per log probability function evaluation. It may be used to calculate predictive inferences as well as to carry out forward simulation for posterior predictive checks; see (Gelman *et al.* 2013) for examples.

The BUGS surgical example explored the ranking of institutions in terms of surgical mortality (Lunn *et al.* 2000, Examples, Volume 1). This is coded in the example in Figure 3 using the generated quantities block. The generated quantity variable `rnk[j]` will hold the rank of institution `j` from 1 to `J` in terms of mortality rate `theta[j]`. The ranks are extracted using the `rank` function. The posterior summary will print average rank and deviation. (Lunn *et al.* 2000) illustrated posterior inference by plotting posterior rank histograms.

Posterior comparisons can be carried out directly or using rankings. For instance, the model in Figure 3 sets `highest[j]` to 1 if hospital `j` has the highest estimated mortality rate (for a discussion of multiple comparisons and hierarchical models, see (Gelman, Hill, and Yajima 2012; Efron 2010)).

As a second illustration, the generated quantities block in Figure 3 calculates the (posterior) probability that a given institution is above-average in terms of mortality rate. This is done for each institution `j` with the usual plug-in estimate of `theta[j] > mean(theta)`, which returns a binary (0 or 1) value. The posterior mean of `above_avg[j]` calculates the posterior probability $\Pr[\theta_j > \bar{\theta}|y, n]$ according to the model.

4.7. Initialization

Stan's samplers and optimizers all start from either random or user-supplied values for each parameter. User supplied initial values are validated and transformed to the underlying unconstrained space; if a parameter value does not satisfy its declared constraints, the program exits and an informative error message is printed. If random initialization is specified, the built-in pseudorandom number generator is called once per unconstrained variable dimension. The default initialization is to randomly generate values uniformly on $[-2, 2]$; another interval may be specified with `init=x` for some non-negative floating-point value `x`. This supplies fairly diffuse starting points when transformed back to the constrained scale, and thus help with convergence diagnostics as discussed in (Gelman *et al.* 2013). Models with more data or more elaborate structure require narrower intervals for initialization to ensure the sampler is able to quickly converge to a stationary distribution in the high mass region of the posterior.

Although Stan is quite effective at converging from diffuse random initializations, the user may supply their own initial values for sampling, optimization, or diagnosis. The top-level command-line option is `init=path`, where `path` is a path to a file specifying values for all parameters in R dump format.

¹²Improper posteriors are diagnosed automatically when parameters overflow to infinity during simulation.

5. Statements

5.1. Assignment and sampling

Stan supports the same two basic statements as BUGS, assignment and sampling, examples of which were introduced earlier. In BUGS, these two kinds of statement define a directed acyclic graphical model; in Stan, they define a log probability function.

Log probability accumulator

There is an implicitly defined variable `lp__` (available in the transformed parameters and model blocks) denoting the log probability that will be returned by the log probability function.

Sampling statements

A sampling statement is nothing more than shorthand for incrementing the log probability accumulator `lp__`. For example, if `beta` is a parameter of type `real`, the sampling statement

```
beta ~ normal(0,1);
```

has the exact same effect (up to dropping constant terms) as the special log probability increment statement

```
increment_log_prob(normal_log(beta,0,1));
```

Define variables before sampling statements

The translation of sampling statements to log probability function evaluations explains why variables must be defined *before* they are used. In particular, a sampling statement does *not* sample the left-hand side variable from the right-hand side distribution.

Parameters are all defined externally by the sampler; all other variables must be explicitly defined with an assignment statement before being used.

Direct definition of probability functions

Because computation is only up to a proportionality constant (an additive constant on the log scale), this sampling statement in turn has the same effect as the direct implementation in terms of basic arithmetic,

```
increment_log_prob(-0.5 * beta * beta);
```

If `beta` is of type `vector`, replace `beta * beta` with `beta' * beta`. Distributions whose probability functions are not built directly into Stan can be implemented directly in this fashion.

5.2. Sequences of statements and execution order

Stan allows sequences of statements wherever statements may occur. Unlike BUGS, in which statements define a directed acyclic graph, in Stan, statements are executed imperatively in the order in which they occur in a program.

Blocks and variable scope

Sequences of statements surrounded by curly braces (`{` and `}`) form blocks. Blocks may start with local variable declarations. The scope of a local variable (i.e., where it is available to be used) is that of the block in which it is declared.

Other variables, such as those declared as data or parameters, may only be assigned to in the block in which they are declared. They may be used in the block in which they are declared and may also be used in any block after the block in which they are declared.

5.3. Whitespace, semicolons, and comments

Following the convention of C++, statements are separated with semicolons in Stan so that the content of whitespace (outside of comments) is irrelevant. This is in contrast to BUGS and R, in which carriage returns are special and may indicate the end of a statement.

Stan supports the line comment style of C++, using two forward slashes (`//`) to comment out the rest of a line; this is the one location where the content of whitespace matters. Stan also supports the line comment style of R and BUGS, treating a pound sign (`#`) as commenting out everything until the end of the line. Stan also supports C++-style block comments, with everything between the start-comment (`/*`) and end-comment (`*/`) markers being ignored.

The preferred style follows that of C++, with line comment used for everything but multiline comments.

Stan follows the C++ convention of separating words in variable names using underbars (`_`), rather than dots (`.`), as used in R and BUGS, or camel case as used in Java.

5.4. Control structures

Stan supports the same kind of explicitly bounded for loops as found in BUGS and R. Like R, but unlike BUGS, Stan supports while loops and conditional (if-then-else) statements.¹³ Stan provides the usual comparison operators and boolean operators to help define conditionals and condition-controlled while loops.

5.5. Print statements and debugging

Stan provides print statements which take arbitrarily many arguments consisting of expressions or string literals consisting of sequences of characters surrounded by double quotes (`"`). These statements may be used for debugging purposes to report on intermediate states of variables or to indicate how far execution has proceeded before an error.

As an example, suppose a user's program raises an error at run time because a covariance matrix defined in the transformed parameters block fails its symmetry constraint.

```
transformed parameters {
  cov_matrix[K] Sigma;
  for (m in 1:M)
    for (n in m:M)
      Sigma[m,n] <- Omega[m,n] * sigma[m] * sigma[n];
```

¹³BUGS omits these control structures because they would introduce ambiguities into the directed, acyclic graph defined by model.

```
    print("Sigma=", Sigma);
}
```

The print statement added at the last line will dump out the values in the matrix.

6. Function and distribution library

In order to support the algorithmic differentiation required to calculate gradients, Hessians, and higher-order derivatives in Stan, we require C++ functions that are templated separately on all of their arguments. In order for these functions to be efficient in computing both values and derivatives, they need to operate directly on vectors of arguments so that shared computations can be reused. For example, if `y` is a vector and `sigma` is a scalar, the logarithm of `sigma` need only be evaluated once in order to compute the normal density for every member of `y` in

```
y ~ normal(mu,sigma);
```

6.1. Basic operators

Stan supports all of the basic C++ arithmetic operators, boolean operators, comparison operators. In addition, it extends the arithmetic operators to matrices and includes pointwise matrix operators.¹⁴ The full set of operators is listed in Figure 4.

6.2. Special functions

Stan provides an especially rich set of special functions. This includes all of the C++ math library functions, as well as numerous more specialized functions such as Bessel functions, gamma and digamma functions, and generalized linear model link functions and their inverses. There are also many compound functions, such as `log1m(x)`, which is more stable arithmetically for values of `x` near 0 than `log(1 - x)`. Stan's special functions are listed in Figure 7 and Figure 8.

In addition to special functions, Stan includes distributions with alternative parameterizations, such as `bernoulli_logit`, which takes a parameter on the log odds (i.e., logit) scale. This allows a more concise notation for generalized linear models as well as more efficient and arithmetically stable execution.

6.3. Matrix and linear algebra functions

Rows, columns and subblocks of matrices can be accessed using `row`, `col`, and `block` functions. Slices of arrays can be accessed using the `head`, `tail`, and `segment` functions. There are also special functions for creating a diagonal matrix from a vector and accessing the diagonal of a vector.

¹⁴This is in contrast to R and BUGS, who treat the basic multiplication and division operators pointwise and use special symbols for matrix operations.

Various reductions are provided for arrays and matrices, such as sums, means, standard deviations, and norms. Replications are also available to copy a value into every cell of a matrix. Slices of matrices and vectors may be accessed by row, column, or general sub-block operations.

Matrix operators use the types of their operands to determine the type of the result. For instance, multiplying a vector by a (column) row vector returns a matrix, whereas multiplying a row vector by a (column) vector returns a real. A postfix apostrophe (') is used for matrix and vector transposition. For example, if y and μ are vectors and Σ is a square matrix, all of the same dimensionality, then $y - \mu$ is a vector, $(y - \mu)'$ is a row vector, $(y - \mu)' * \Sigma$ is a row vector, and $(y - \mu)' * \Sigma * (y - \mu)$ will be a real value. Matrix division is provided, which is much more arithmetically stable than inversion, e.g., $(y - \mu)' / \Sigma$ computes the same function as $(y - \mu)' * \text{inverse}(\Sigma)$. Stan also supports elementwise multiplication ($.*$) and division ($./$).

Linear algebra functions are provided for trace, left and right division, Cholesky factorization, determinants and log determinants, inverses, eigenvalues and eigenvectors, and singular value decomposition. All of these operations may be applied to matrices of parameters or constants. Various functions are specialized for speed, such as quadratic products, diagonal specializations, and multiply by self transposed; e.g., the previous example $(y - \mu)' * \Sigma * (y - \mu)$ could be coded as `quad_form(Sigma, y - mu)`.

The full set of matrix and linear-algebra functions is listed in Figure 9; operators, which also apply to matrices and vectors, are listed in Figure 4.

6.4. Probability functions

Stan supports a growing collection of built-in univariate and multivariate probability density and mass functions. These probability functions share various features of their declarations and behavior.

All probability functions are defined on the log scale to avoid underflow. They are all named with the suffix `_log`, e.g., `normal_log()`, is the log-scale normal distribution density function.

All probability functions check that their arguments are within the appropriate constrained support and are configured to throw exceptions and print error messages for out-of-domain arguments (the behavior of positive and negative infinity and not-a-number values are built into floating-point arithmetic). For example, `normal_log(y, mu, sigma)` requires the scale parameter `sigma` to be non-negative. Exceptions that are raised by functions will be caught by the sampler, optimizer or diagnostic, and their warning messages will be printed for the user. Log density evaluations in which exceptions are raised are treated as if they had evaluated to negative infinity, and are thus rejected by the sampler or optimizer.

The list of probability functions is provided in Figure 10, Figure 11, and Figure 12.

Up to a proportion calculations

All probability functions support calculating results up to a constant proportion, which becomes an additive constant on the log scale. Constancy here refers to being a numeric literal such as 1 or 0.5, a constant function such as `pi()`, data and transformed data variables, or a function that only depends on literals, constant functions or data variables.

Non-constants include parameters, transformed parameters, local variables declared in the

transformed parameters or model statements, as well as any expression involving a non-constant.

Constant terms are dropped from probability function calculations at the time the model is compiled, so there is no run-time overhead to decide which expressions denote constants.¹⁵ For example, executing `y ~ normal(0, sigma)` only evaluates `log(sigma)` if `sigma` is a parameter, transformed parameter, or a local variable in the transformed parameters or model block; that is, `log(sigma)` is not evaluated if `sigma` is constant as defined above.

Constant terms are *not* dropped in explicit function evaluations, such as `normal_log(y, 0, sigma)`.

Vector arguments and shared computations

All of the univariate probability functions in **Stan** are implemented so that they accept arrays or vectors of arguments. For example, although the basic signature of the probability function `normal_log(y, mu, sigma)` involves real `y`, `mu` and `sigma`, it supports calls in which any or all of `y`, `mu` and `sigma` contain more than one element. A typical use case would be for linear regression, such as `y ~ normal(X * beta, sigma)`, where `y` is a vector of observed data, `X` is a predictor matrix, `beta` is a coefficient vector, and `sigma` is a real value for the noise scale.

The advantage of using vectors is twofold. First, the models are more concise and closer to mathematical notation. Second, the vectorized versions are much faster. They reduce the number of times expensive operations need to be evaluated and also reduce the number of virtual function calls required in the compiled C++ executable for calculating gradients and higher-order derivatives. For example if `sigma` is a parameter, then in evaluating `y` `normal(X * beta, sigma)`, the logarithm of `sigma` need only be computed once; if either `y` or `beta` is an N -vector, it also reduces the number of virtual function calls from N to 1.

7. Built-in inference engines

Stan includes several Markov chain Monte Carlo (MCMC) samplers and several optimizers. Others may be straightforwardly implemented within **Stan**'s C++ framework for sampling and optimization using the log probability and derivative information supplied by a model.

7.1. Markov chain Monte Carlo samplers

Hamiltonian Monte Carlo

The MCMC samplers provided include Euclidean Hamiltonian Monte Carlo (EHMC, which in much of the literature is referenced as simply HMC) (Duane, Kennedy, Pendleton, and Roweth 1987; Neal 1994, 2011) and the no-U-turn sampler (NUTS) (Hoffman and Gelman 2014). Both the basic and NUTS versions of HMC allow estimation or specification of unit, diagonal, or full mass matrices. NUTS, the default sampler for **Stan**, automatically adapts the number of leapfrog steps, eliminating the need for user-specified tuning parameters. Both algorithms take advantage of gradient information in the log probability function to generate

¹⁵Both vector arguments and dropping constant terms are implemented in C++ through template metaprograms that infer traits of template arguments to the probability functions. Whether to drop constants is configurable through a boolean template parameter on the log probability and derivative functions generated in C++ for a model.

coherent motion through the posterior that dramatically reduces the autocorrelation of the resulting transitions.

7.2. Optimizers

In addition to performing full Bayesian inference via posterior sampling, *Stan* also can perform optimization (i.e., computation of the posterior mode). We are currently working on implementing other optimization-based inference approaches including variational Bayes, expectation propagation, and marginal inference using approximate integration. All these algorithms require optimization steps.

BFGS

The default optimizer in *Stan* is the Broyden-Fletcher-Goldfarb-Shanno (BFGS) optimizer. BFGS is a quasi-Newton optimizer that evaluates gradients directly, then uses the gradients to update an approximation to the Hessian. Plans are in the works to also include the more involved, but more scalable limited-memory BFGS (L-BFGS) scheme. [Nocedal and Wright \(2006\)](#) cover both BFGS and L-BFGS samplers.

Conjugate gradient

Stan provides a standard form of conjugate gradient optimization; see [\(Nocedal and Wright 2006\)](#). As its name implies, conjugate gradient optimization requires gradient evaluations.

Accelerated gradient

Additionally, *Stan* implements a crude version of Nesterov’s accelerated gradient optimizer [Nesterov \(1983\)](#), which combines gradient updates with a momentum-like update to hasten convergence.

Acknowledgments

First and foremost, we would like to thank all of the users of *Stan* for taking a chance on a new package and sticking with it as we ironed out the details in the first release. We’d like to particularly single out the students in Andrew Gelman’s Bayesian data analysis courses at Columbia University and Harvard University, who served as trial subjects for both *Stan* and [\(Gelman et al. 2013\)](#).

We’d like to thank Aki Vehtari for comments and corrections on a draft of the paper.

Stan was and continues to be supported largely through grants from the U. S. government. Grants which indirectly supported the initial research and development included grants from the Department of Energy (DE-SC0002099), the National Science Foundation (ATM-0934516), and the Department of Education Institute of Education Sciences (ED-GRANTS-032309-005 and R305D090006-09A). The high-performance computing facility on which we ran evaluations was made possible through a grant from the National Institutes of Health (1G20RR030893-01). *Stan* is currently supported in part by a grant from the National Science Foundation (CNS-1205516).

We would like to thank those who have contributed code for new features, Jeffrey Arnold,

Operation	Precedence	Associativity	Placement	Description
<code> </code>	9	left	binary infix	logical or
<code>&&</code>	8	left	binary infix	logical and
<code>==</code>	7	left	binary infix	equality
<code>!=</code>	7	left	binary infix	inequality
<code><</code>	6	left	binary infix	less than
<code><=</code>	6	left	binary infix	less than or equal
<code>></code>	6	left	binary infix	greater than
<code>>=</code>	6	left	binary infix	greater than or equal
<code>+</code>	5	left	binary infix	addition
<code>-</code>	5	left	binary infix	subtraction
<code>*</code>	4	left	binary infix	multiplication
<code>/</code>	4	left	binary infix	(right) division
<code>\</code>	3	left	binary infix	left division
<code>.*</code>	2	left	binary infix	elementwise multiplication
<code>./</code>	2	left	binary infix	elementwise division
<code>!</code>	1	n/a	unary prefix	logical negation
<code>-</code>	1	n/a	unary prefix	negation
<code>+</code>	1	n/a	unary prefix	promotion (no-op in Stan)
<code>'</code>	0	n/a	unary postfix	transposition
<code>()</code>	0	n/a	prefix, wrap	function application
<code>[]</code>	0	left	prefix, wrap	array, matrix indexing

Figure 4: Each of Stan’s unary and binary operators follow strict precedences, associativities, placement within an expression. The operators are listed in order of precedence, from least tightly binded to most tightly binding.

Function	Description
<code>e</code>	base of natural logarithm
<code>epsilon</code>	smallest positive number
<code>negative_epsilon</code>	largest negative value
<code>negative_infinity</code>	negative infinity
<code>not_a_number</code>	not-a-number
<code>pi</code>	π
<code>positive_infinity</code>	positive infinity
<code>sqrt2</code>	square root of two

Figure 5: Stan implements a variety of useful constants.

Yuanjun Gao, and Marco Inacio, as well as those who contributed documentation corrections and code patches, Jeffrey Arnold, David R. Blair, Eric C. Brown, Eric N. Brown, Devin Caughey, Wayne Folta, Andrew Hunter, Marco Inacio, Louis Luangkesorn, Jeffrey Oldham, Mike Ross, Terrance Savitsky, Yajuan Si, Dan Stowell, Zhenming Su, and Dougal Sutherland.

Finally, we would like to thank two anonymous referees.

Function	Description
<code>acos</code>	arc cosine
<code>acosh</code>	arc hyperbolic cosine
<code>asin</code>	arc sine
<code>asinh</code>	arc hyperbolic sine
<code>atan</code>	arc tangent
<code>atan2</code>	arc ratio tangent
<code>atanh</code>	arc hyperbolic tangent
<code>cos</code>	cosine
<code>cosh</code>	hyperbolic cosine
<code>hypot</code>	hypoteneuse
<code>sin</code>	sine
<code>sinh</code>	hyperbolic sine
<code>tan</code>	tangent
<code>tanh</code>	hyperbolic tangent

Figure 6: *Stan implements both circular and hyperbolic trigonometric functions, as well as their inverses.*

A. Developer process

A.1. Version control and source repository

Stan’s source code is hosted on GitHub and managed using the **Git** version control system (Chacon 2009). To manage the workflow with so many developers working at any given time, the project follows the GitFlow process (Driessen 2010). All developer submissions are managed through pull requests and we have gratefully received patches from numerous sources outside the core development team.

A.2. Continuous integration

Stan uses continuous integration, meaning that the entire program and set of tests are run automatically as code is pushed to the **Git** repository. Each pull request is tested for compatibility with the development branch, and the development branch itself is tested for stability. Stan uses Jenkins (Smart 2011), an open-source continuous integration server.

A.3. Testing framework

Stan includes extensive unit tests for low-level C++ code. Unit tests are implemented using the **googletest** framework (Google 2011). The probability functions and command-line invocations are complex enough that programs are used to automatically generate test code for **googletest**. These unit tests evaluate every function for both appropriate values and appropriate derivatives. This requires an extensive meta-testing framework for the probability distributions due to their high degree of configurability as to argument types. The testing portion of the makefile also runs tests of all of the built-in models, including almost all of the BUGS sample models. Models are tested for both convergence and posterior mean estimation to within

Function	Description
<code>abs</code>	double absolute value
<code>abs</code>	integer absolute value
<code>binary_log_loss</code>	log loss
<code>bessel_first_kind</code>	Bessel function of the first kind
<code>bessel_second_kind</code>	Bessel function of the second kind
<code>binomial_coefficient_log</code>	log binomial coefficient
<code>cbrt</code>	cube root
<code>ceil</code>	ceiling
<code>cumulative_sum</code>	cumulative sum
<code>erf</code>	error function
<code>erfc</code>	complementary error function
<code>exp</code>	base- e exponential
<code>exp2</code>	base-2 exponential
<code>expm1</code>	exponential of quantity minus one
<code>fabs</code>	real absolute value
<code>fdim</code>	positive difference
<code>floor</code>	floor
<code>fma</code>	fused multiply-add
<code>fmax</code>	floating-point maximum
<code>fmin</code>	floating-point minimum
<code>fmod</code>	floating-point modulus
<code>if_else</code>	conditional
<code>int_step</code>	Heaviside step function
<code>inv</code>	inverse (one over argument)
<code>inv_cloglog</code>	inverse of complementary log-log
<code>inv_logit</code>	logistic sigmoid
<code>inv_sqrt</code>	inverse square root
<code>inv_square</code>	inverse square
<code>lbeta</code>	log beta function
<code>lgamma</code>	log Γ function
<code>lmgamma</code>	log multi- Γ function
<code>log</code>	natural (base- e) logarithm
<code>log10</code>	base-10 logarithm
<code>log1m</code>	natural logarithm of one minus argument
<code>log1m_exp</code>	natural logarithm of one minus natural exponential
<code>log1m_inv_logit</code>	natural logarithm of logistic sigmoid
<code>log1p</code>	natural logarithm of one plus argument
<code>log1p_exp</code>	natural logarithm of one plus natural exponential
<code>log2</code>	base-2 logarithm

Figure 7: *Stan implements many special and transcendental functions.*

Function	Description
<code>log_diff_exp</code>	natural logarithm of difference of exponentials
<code>log_falling_factorial</code>	falling factorial (Pochhammer)
<code>log_inv_logit</code>	natural logarithm of the logistic sigmoid
<code>log_rising_factorial</code>	falling factorial (Pochhammer)
<code>log_sum_exp</code>	logarithm of the sum of exponentials of arguments
<code>logit</code>	log-odds
<code>max</code>	integer maximum
<code>max</code>	real maximum
<code>mean</code>	sample average
<code>min</code>	integer minimum
<code>min</code>	real minimum
<code>modified_bessel_first_kind</code>	modified Bessel function of the first kind
<code>modified_bessel_second_kind</code>	modified Bessel function of the second kind
<code>multiply_log</code>	multiply linear by log
<code>owens_t</code>	Owens-t
<code>phi</code>	Φ function (cumulative unit normal)
<code>phi_approx</code>	efficient, approximate Φ
<code>pow</code>	power (i.e., exponentiation)
<code>prod</code>	product of sequence
<code>rank</code>	rank of element in array or vector
<code>rep_array</code>	fill array with value
<code>round</code>	round to nearest integer
<code>sd</code>	sample standard deviation
<code>softmax</code>	softmax (multi-logit link)
<code>sort_asc</code>	sort in ascending order
<code>sort_desc</code>	sort in descending order
<code>sqrt</code>	square root
<code>square</code>	square
<code>step</code>	Heaviside step function
<code>sum</code>	sum of sequence
<code>tgamma</code>	Γ function
<code>trunc</code>	truncate real to integer
<code>variance</code>	sample variance

Figure 8: *Special functions (continued).*

Function	Description
block	sub-block of matrix
cholesky_decompose	Cholesky decomposition
col	column of matrix
cols	number of columns in matrix
columns_dot_product	dot product of matrix columns
columns_dot_self	dot product of matrix columns with self
crossprod	cross-product
determinant	matrix determinant
diag_matrix	vector to diagonal matrix
diag_post_multiply	post-multiply matrix by diagonal matrix
diag_pre_multiply	pre-multiply matrix by diagonal matrix
diagonal	diagonal of matrix as vector
dims	dimensions of matrix
dot_product	dot product
dot_self	dot product with self
eigenvalues_sym	eigenvalues of symmetric matrix
eigenvectors_sym	eigenvectors of symmetric matrix
head	head of vector
inverse	matrix inverse
inverse_spd	symmetric, positive-definite matrix inverse
log_determinant	natural logarithm of determinant
mdivide_left_tri_low	lower-triangular matrix left division
mdivide_right_tri_low	lower-triangular matrix right division
multiply_lower_tri_self_transpose	multiply lower-triangular by transpose
quad_form	quadratic form vector-matrix multiplication
rep_matrix	replicate scalar, row vector or vector to matrix
rep_row_vector	replicate scalar to row vector
rep_vector	replicate scalar to vector
row	row of matrix
rows	number of rows in matrix
rows_dot_product	dot-product of rows of matrices
rows_dot_self	dot-product of matrix with itself
segment	sub-vector
singular_values	singular values of matrix
size	number of entries in array or vector
sub_col	sub-column of matrix
sub_row	sub-row of matrix
tail	tail of vector
tcrossprod	matrix post-multiply by own transpose
trace	trace of matrix
trace_gen_quad_form	trace of generalized quadratic form
trace_quad_form	trace of quadratic form

Figure 9: *A large suite of matrix functions admits efficient multivariate model implementation in Stan.*

Function	Description
<code>bernoulli_cdf</code>	Bernoulli cdf
<code>bernoulli_log</code>	log Bernoulli pmf
<code>bernoulli_logit_log</code>	logit-scale log Bernoulli pmf
<code>bernoulli_rng</code>	Bernoulli RNG
<code>beta_binomial_cdf</code>	beta-binomial cdf
<code>beta_binomial_log</code>	log beta-binomial pmf
<code>beta_binomial_rng</code>	beta-binomial rng
<code>beta_cdf</code>	beta cdf
<code>beta_log</code>	log beta pdf
<code>beta_rng</code>	beta RNG
<code>binomial_cdf</code>	binomial cdf n
<code>binomial_log</code>	log binomial pmf
<code>binomial_logit_log</code>	log logit-scaled binomial pmf
<code>binomial_rng</code>	binomial RNG
<code>categorical_log</code>	log categorical pmf
<code>categorical_rng</code>	categorical RNG
<code>cauchy_cdf</code>	Cauchy cdf
<code>cauchy_log</code>	log Cauchy pdf
<code>cauchy_rng</code>	Cauchy RNG
<code>chi_square_log</code>	log chi-square pdf
<code>chi_square_rng</code>	chi-square RNG
<code>dirichlet_log</code>	log Dirichlet pdf
<code>dirichlet_rng</code>	Dirichlet RNG
<code>double_exponential_log</code>	log double-exponential (Laplace) pdf
<code>double_exponential_rng</code>	double-exponential (Laplace) RNG
<code>exp_mod_normal_cdf</code>	exponentially modified normal cdf
<code>exp_mod_normal_log</code>	log exponentially modified normal pdf
<code>exp_mod_normal_rng</code>	exponentially modified normal RNG
<code>exponential_cdf</code>	exponential cdf
<code>exponential_log</code>	log of exponential pdf
<code>exponential_rng</code>	exponential RNG
<code>gamma_log</code>	log gamma pdf
<code>gamma_rng</code>	gamma RNG
<code>gumbel_cdf</code>	Gumbel cdf
<code>gumbel_log</code>	log Gumbel pdf
<code>gumbel_rng</code>	Gumbel RNG
<code>hypergeometric_log</code>	log hypergeometric pmf
<code>hypergeometric_rng</code>	hypergeometric RNG
<code>inv_chi_square_cdf</code>	inverse chi-square cdf
<code>inv_chi_square_log</code>	log inverse chi-square pdf
<code>inv_chi_square_rng</code>	inverse chi-square RNG

Figure 10: *Most common probability distributions have explicitly implemented in Stan.*

Function	Description
<code>inv_gamma_cdf</code>	inverse gamma cdf
<code>inv_gamma_log</code>	log inverse gamma pdf
<code>inv_gamma_rng</code>	inverse gamma RNG
<code>inv_wishart_log</code>	log inverse Wishart pdf
<code>inv_wishart_rng</code>	inverse Wishart RNG
<code>lkj_corr_cholesky_log</code>	log LKJ correlation, Cholesky-variate pdf
<code>lkj_corr_cholesky_rng</code>	LKJ correlation, Cholesky-variate RNG
<code>lkj_corr_log</code>	log of LKJ correlation pdf
<code>lkj_corr_rng</code>	LKJ correlation RNG
<code>logistic_cdf</code>	logistic cdf
<code>logistic_log</code>	log logistic pdf
<code>logistic_rng</code>	logistic RNG
<code>lognormal_cdf</code>	lognormal cdf
<code>lognormal_log</code>	log of lognormal pdf
<code>lognormal_rng</code>	lognormal RNG
<code>multi_normal_cholesky_log</code>	log multi-normal Cholesky-parameterized pdf
<code>multi_normal_log</code>	log multi-normal pdf
<code>multi_normal_prec_log</code>	log multi-normal precision-parameterized pdf
<code>multi_normal_rng</code>	multi-normal RNG
<code>multi_student_t_log</code>	log multi student-t pdf
<code>multi_student_t_rng</code>	multi student-t RNG
<code>multinomial_log</code>	log multinomial pmf
<code>multinomial_rng</code>	multinomial RNG
<code>neg_binomial_cdf</code>	negative binomial cdf
<code>neg_binomial_log</code>	log negative binomial pmf
<code>neg_binomial_rng</code>	negative binomial RNG
<code>normal_cdf</code>	normal cdf
<code>normal_log</code>	log normal pdf (c.f. log lognormal pdf)
<code>normal_rng</code>	normal RNG
<code>ordered_logistic_log</code>	log ordinal logistic pmf
<code>ordered_logistic_rng</code>	ordinal logistic RNG
<code>pareto_cdf</code>	Pareto cdf
<code>pareto_log</code>	log Pareto pdf
<code>pareto_rng</code>	Pareto RNG
<code>poisson_cdf</code>	Poisson cdf
<code>poisson_log</code>	log Poisson pmf
<code>poisson_log_log</code>	log Poisson log-parameter pdf
<code>poisson_rng</code>	Poisson RNG

Figure 11: *Probability functions (continued)*

Function	Description
<code>rayleigh_log</code>	log Rayleigh pdf
<code>rayleigh_rng</code>	Rayleigh RNG
<code>rayleigh_cdf</code>	Rayleigh cdf
<code>scaled_inv_chi_square_cdf</code>	scaled inverse-chi-square cdf
<code>scaled_inv_chi_square_log</code>	log scaled inverse-chi-square pdf
<code>scaled_inv_chi_square_rng</code>	scaled inverse-chi-square RNG
<code>skew_normal_cdf</code>	skew-normal cdf
<code>skew_normal_log</code>	log of skew-normal pdf
<code>skew_normal_rng</code>	skew-normal RNG
<code>student_t_cdf</code>	Student-t cdf
<code>student_t_log</code>	log of Student-t pdf
<code>student_t_rng</code>	Student-t RNG
<code>uniform_log</code>	log of uniform pdf
<code>uniform_rng</code>	uniform RNG
<code>weibull_cdf</code>	Weibull cdf
<code>weibull_log</code>	log of Weibull pdf
<code>weibull_rng</code>	Weibull RNG
<code>wishart_log</code>	log of Wishart pdf
<code>wishart_rng</code>	Wishart RNG

Figure 12: *Probability functions (continued 2)*

MCMC standard error.

A.4. Builds

The build process for Stan is highly automated through a cross-platform series of `make` files. The top-level makefile builds the Stan-to-C++ translator command `bin/stanc` and posterior analysis command `bin/print`. It also builds the library archive `bin/libstan.a`. Great care was taken to avoid complicated platform-dependent configuration requirements that place a high burden on user system knowledge for installation. All that is needed is a relatively recent C++ compiler and version of `make`.

As exemplified in the introduction, the makefile is automated enough to build an executable form of a Stan model in a single command. All libraries and other executables will be built as a side effect.

The top-level makefile also supplies targets to build all of the documentation. C++ API documentation is generated using the `doxygen` package (van Heesch 2011). The Stan manual (Stan Development Team 2014) is typeset using the \LaTeX package (Mittelbach, Goossens, Braams, Carlisle, and Rowley 2004).

The makefile also has targets for all of the unit and functional testing, for building the source-level distribution, and for cleaning any temporary files that it creates.

B. Library dependencies

Stan’s modeling language is only dependent on two external libraries.

B.1. Boost

Stan depends on several of the **Boost** C++ libraries (Schäling 2011). Stan makes extensive use of **Boost**’s template metaprogramming facilities including the **Enable if** package, the **Type Traits** library, and the **Lexical Cast** library. The Stan language is parsed using **Boost**’s **Spirit** parser, which itself depends on the binding libraries **Phoenix**, **Bind**, and **Lambda**, the variant type library **Variant**, and the container library **Fusion**. Exceptions are largely handled and configured through the error checking facilities in the **Math** and **Exception** packages. Output formatting and ad-hoc input parsing for various formats is facilitated with the **Format** library. Stan relies heavily on the special functions defined in the **Math** subpackages **Special Functions** and **Statistical Distributions**. Random number generation is carried out using the **Random** package. The posterior analysis framework and some built-in functions depend on the **Accumulators** package.

B.2. Eigen

Stan’s handling of matrices and linear algebra is implemented through the **Eigen** C++ template library (Guennebaud and Jacob 2012). Eigen uses template metaprogramming to achieve state-of-the-art performance for matrix and linear algebra operations with a great deal of flexibility with respect to input types. Unfortunately, many of the expression templates that Eigen uses for efficient static analysis and lazy evaluation are short-circuited because of Stan’s need to have mixed type operations (i.e., multiplying a constant predictor matrix of double values by a parameter vector of algorithmic differentiation values). To make up for this in some important cases, Stan has provided compound functions such as the quadratic form, which allow speedups of both the matrix operations and their derivatives compared to a direct implementation using Stan’s built-in operators.

References

- Chacon S (2009). *Pro Git*. Apress. ISBN 978-1-4302-1833-3.
- Driessen V (2010). “A Successful Git Branching Model.” URL <http://nvie.com/posts/a-successful-git-branching-model/>.
- Duane A, Kennedy A, Pendleton B, Roweth D (1987). “Hybrid Monte Carlo.” *Physics Letters B*, **195**(2), 216–222.
- Efron B (2010). *Large-Scale Inference: Empirical Bayes Methods for Estimation, Testing, and Prediction*. Institute of Mathematical Statistics Monographs. Cambridge University Press.
- Gelman A, Carlin JB, Stern HS, Dunson DB, Vehtari A, Rubin DB (2013). *Bayesian Data Analysis*. Third edition. CRC/Chapman-Hall, London.
- Gelman A, Hill J, Yajima M (2012). “Why We (Usually) Don’t Have to Worry about Multiple Comparisons.” *Journal of Research on Educational Effectiveness*, **5**, 189–211.
- Gelman A, Rubin DB (1992). “Inference from Iterative Simulation Using Multiple Sequences.” *Statistical Science*, **7**(4), 457–472. ISSN 0883-4237.
- Geyer CJ (2011). “Introduction to Markov Chain Monte Carlo.” In *Handbook of Markov Chain Monte Carlo*. Chapman & Hall/CRC.
- Google (2011). “Google Test: Google C++ Testing Framework.” <http://code.google.com/p/googletest/>.
- Guennebaud G, Jacob B (2012). “Eigen C++ Library, Version 3.1.” <http://eigen.tuxfamily.org/>.
- Hoffman MD, Gelman A (2014). “The No-U-Turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo.” *Journal of Machine Learning Research*, **15**(Apr), 1593–1623.
- Lunn D, Jackson C, Best N, Thomas A, Spiegelhalter D (2012). *The BUGS Book—A Practical Introduction to Bayesian Analysis*. CRC Press / Chapman and Hall.
- Lunn D, Spiegelhalter D, Thomas A, Best N (2009). “The BUGS project: Evolution, Critique, and Future Directions.” *Statistics in Medicine*, **28**, 3049–3067. URL <http://www.openbugs.info/w/>.
- Lunn D, Thomas A, Best N, Spiegelhalter D (2000). “WinBUGS—A Bayesian Modelling Framework: Concepts, Structure, and Extensibility.” *Statistics and Computing*, **10**, 325–337. URL www.mrc-bsu.cam.ac.uk/bugs/.
- Metropolis N, Rosenbluth AW, Rosenbluth MN, Teller A, Teller E (1953). “Equation of State Calculations by Fast Computing Machines.” *Journal of Chemical Physics*, **21**, 1087–1092.
- Mittelbach F, Goossens M, Braams J, Carlisle D, Rowley C (2004). *The LaTeX Companion*. Tools and Techniques for Computer Typesetting, 2nd edition. Addison-Wesley.

- Neal R (2011). “MCMC Using Hamiltonian Dynamics.” In S Brooks, A Gelman, GL Jones, XL Meng (eds.), *Handbook of Markov Chain Monte Carlo*, pp. 116–162. Chapman and Hall/CRC.
- Neal RM (1994). “An Improved Acceptance Procedure for the Hybrid Monte Carlo Algorithm.” *Journal of Computational Physics*, **111**, 194–203.
- Neal RM (2003). “Slice Sampling.” *Annals of Statistics*, **31**(3), 705–767.
- Nesterov Y (1983). “A Method Of Solving A Convex Programming Problem with Convergence Rate $O(1/K^2)$.” *Soviet Mathematics Doklady*, **27**(2), 372–376.
- Nocedal J, Wright SJ (2006). *Numerical Optimization*. Second edition. Springer-Verlag. ISBN 978-0-387-30303-1.
- 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*. Vienna, Austria. URL [www-fis.iarc.fr/\\$\sim\\$martyn/software/jags/](http://www-fis.iarc.fr/\simmartyn/software/jags/).
- Plummer M, Best N, Cowles K, Vines K (2006). “CODA: Convergence Diagnosis and Output Analysis for MCMC.” *R News*, **6**(1), 7–11. URL <http://CRAN.R-project.org/doc/Rnews/>.
- R Core Team (2014). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL <http://www.R-project.org>.
- Schäling B (2011). *The Boost C++ Libraries*. XML Press. URL <http://www.boost.org/>.
- Smart JF (2011). *Jenkins: The Definitive Guide*. O’Reilly Media.
- Stan Development Team (2014). *Stan Modeling Language Users Guide and Reference Manual*. Version 2.1.0, URL <http://mc-stan.org/manual.html>.
- van Heesch D (2011). “Doxygen: Generate Documentation from Source Code.” <http://www.stack.nl/~dimitri/doxygen/index.html>.

Affiliation:

Bob Carpenter
 Department of Statistics
 Columbia University
 1255 Amsterdam Avenue
 New York, NY 10027
 U.S.A.
 E-mail: carp@alias-i.com
 URL: <http://mc-stan.org/>

Journal of Statistical Software

published by the American Statistical Association

Volume VV, Issue II
 MMMMMM YYYY

<http://www.jstatsoft.org/>

<http://www.amstat.org/>

Submitted: yyyy-mm-dd

Accepted: yyyy-mm-dd
