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 Univesity

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 2.1, Stan provides full Bayesian inference for continuous-variable models through Markov chain Monte Carlo (MCMC) methods such as the No-U-Turn sampler (NUTS), an adaptive form of Hamiltonian Monte Carlo (HMC) sampling. Penalized maximum likelihood estimates are calculated using optimization methods such as BFGS.

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 I/O transforms.

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

1. Why Stan?

We did not set out to build Stan as it currently exists.[1] Our original goal was to apply full Bayesian inference to the sort of multilevel generalized linear models discussed in Part II of (Gelman and Hill 2007), which are structured with grouped and interacted predictors at multiple levels, hierarchical covariance priors, nonconjugate coefficient priors, latent effects as in item-response models, and varying output link functions and distributions.

These models turned out to be a challenge for existing general purpose inference software. A direct encoding in BUGS (Lunn, Thomas, and Spiegelhalter 2000; Lunn, Spiegelhalter, Thomas, and Best 2009; Lunn, Jackson, Best, Thomas, and Spiegelhalter 2012) or JAGS (Plummer 2003) can grind these tools to a halt. We began by adding custom vectorized logistic regressions to JAGS using C++ to overcome the cost of interpretation. Although this is much faster than looping in JAGS, it quickly became clear that the root of the problem was the slow convergence of conditional sampling when parameters were highly correlated in the posterior, as for time-series models and hierarchical models with interacted predictors. We finally realized we needed a better sampler, not a more efficient implementation of Gibbs sampling.

We briefly considered trying to tune proposals for a random-walk Metropolis-Hastings sampler, but that seemed too problem specific and not even necessarily possible without some kind of adaptation rather than a global tuning of the proposals.

We were hearing more about Hamiltonian Monte Carlo (HMC) sampling, which appeared promising but was also problematic in that the Hamiltonian dynamics simulation requires the gradient of the log posterior. Although not difficult in theory for most functions, computing these gradients by hand on a model-by-model basis is very tedious and error prone. That is when we discovered reverse-mode algorithmic differentiation, which, given a templated C++ function for the log posterior, automatically computes its analytic gradient up to machine precision accuracy in only a few multiples of the time to evaluate the log probability function itself.[2] We explored existing algorithmic differentiation packages with open licenses such as **RAD** (Gay 2005) and its repackaging in the Sacado module of the **Trilinos** toolkit, and the **CppAD** package of the COIN-OR toolkit (Bell and Burke 2008). Both packages, however, supported few special functions (e.g., probability functions, log gamma, inverse logit) or linear algebra operations (e.g., Cholesky decompositions, matrix division), and neither are easily or modularly extensible.

Consequently we built our own reverse-mode algorithmic differentiation package. At that point, we ran into the problem that we could not just plug in the probability functions from a package like **Boost** because they weren't templated generally enough across all arguments. Rather than pay the price of promoting floating point values to algorithmic differentiation variables, we wrote our own fully templated probability functions and other special functions.

Next, we integrated the C++ package **Eigen** for matrix operations and linear algebra functions. **Eigen** makes extensive use of expression templates for lazy evaluation and the curiously

[1] Stan's home page, http://mc-stan.org/, links to the Stan manual, example models including translations of the BUGS examples and the models from (Gelman and Hill 2007), getting started instructions and full documentation for Stan's interfaces for the command line shell (**CmdStan**), Python (**PyStan**), and R (**RStan**), and the source code repositories and issue trackers.

[2] Hessian matrices (i.e., all second order derivatives) are more expensive to calculate; each row corresponds to a parameter and is filled by the gradient of the derivative of the log probability function with respect to the parameter.

recurring template pattern (CRTP) to implement concepts without virtual function calls; Vandevoorde and Josuttis (2002) provide a complete description of template metaprogramming techniques. Unfortunately, we ran into the same problem with Eigen as with the existing probability libraries—it doesn't support mixed operations of algorithmic differentiation variables and primitives like `double`. Although we initially began by promoting floating-point vectors to algorithmic differentiation variables, we later completely rewrote mixed-mode operations like multiplying a data matrix by a parameter vector. To compute derivatives of matrix operations such as division and inverse, we implemented specialized derivatives using the rules described by Giles (2008).

At this point, we could fit models coded directly in C++ on top of the pre-release versions of the Stan application programming interface (API). Seeing how well this all worked, we set our sights on the generality and ease of use of BUGS and designed a modeling language in which statisticians could write their models in familiar notation that could be transformed to efficient C++ code before being compiled into an efficient executable program. Although we started with the target of specifying directed graphical models, our first implementation turned out to be a more general imperative form of specifying log probability functions; BUGS models could be translated line for line, but the Stan language also supports much richer imperative constructs like conditionals, while loops, and local variables. This paper is primarily about the Stan language and how it can be used to specify statistical models.

The next problem we ran into when we started implementing richer models was variables with constrained support, such as positive variables, simplexes, and covariance matrices. Efficient implementation of these constraints required the introduction of typed variables which automatically transformed to unconstrained support with suitable adjustments to the log probability from the log absolute Jacobian determinant of the inverse transforms.

Even with the prototype compiler generating models, we still faced a major hurdle for ease of use. The efficiency of HMC is very sensitive to two tuning parameters, the discretization interval (i.e., step size) and the total simulation time (i.e., number of steps). The interval size parameter could be tuned during warm-up based on Metropolis rejection rates, but the number of steps proved difficult to tune without sacrificing the detailed balance of the sampler. This led to the development of the No U-Turn (NUTS) sampler (Hoffman and Gelman 2011). Roughly speaking, NUTS builds a tree of possible samples by randomly simulating Hamiltonian dynamics both forwards and backwards in time until the combined trajectory turns back on itself. Once the trajectory has terminated, a new sample is drawn from the tree. In its original version, NUTS also estimates a step size during warmup based on a target acceptance probability.

We thought we were home free at this point, but when we measured the speed of some BUGS examples versus Stan we were very disappointed. BUGS's very first example model, Rats, ran more than an order of magnitude faster in JAGS than in Stan. Rats is a tough test case because the conjugate priors and lack of posterior correlations make it an ideal candidate for efficient Gibbs sampling. Realizing that we were doing redundant calculations, we wrote a vectorized form of the normal distribution for multiple variates with the same mean and scale, which sped things up a bit. Performance was lacking until we finally figured out how to both eliminate redundant calculations and partially evaluate the gradients using a combination of expression templates and metaprogramming.

When we attempted to fit a time-series model, we found that normalizing the data to unit

```
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.*

sample mean and variance sped up the fits by an order of magnitude. In hindsight this isn't surprising, as the performance of the numerical simulation in HMC scales with the variation of the parameter scales. In Stan 1.0 we introduced an adaptive diagonal metric (mass matrix) into our HMC implementations that allowed the parameter scales to be standardized automatically; Stan 2.0 added an option to estimate a dense metric (mass matrix) and a full covariance standardization of the posterior.

Because the diagonal and dense metrics perform only a global standardization, the performance of Stan in models where the relative parameter scales varies by location, such as hierarchical and latent models, can suffer (hierarchical modeling is covered in Sections 3.1 and reparameterizations to speed it up in 3.11). Riemannian manifold Hamiltonian Monte Carlo (RHMC) introduces a location-dependent metric that can overcome these final hurdles (Girolami and Calderhead 2011); Stan has a prototype implementation of RHMC based on the SoftAbs metric of (Betancourt 2012), using a generalization of NUTS to Riemannian manifolds (Betancourt 2013).

2. Overview

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.[3] The model treats the observed binary data, `y[1],...,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:

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

```
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.[4] For example, 10 observations for the model in Figure 1 could be encoded as[5]

```
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. Compling the model

After a C++ compiler and `make` are installed,[6] 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.[7]

```
> 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

[4] A **JSON** interface for structured data input and output is currently under development.

[5] This data file is provided with the Stan distrbution in file `src/models/basic_estimators/bernoulli.R.stan`.

[6] Appropriate 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.

[7] Before 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.

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.

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.

```
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. These parameters may be set on the command line.

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 2013).

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 = 0.050000000000000003 (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 = 847896134
# output
#   file = output.csv (Default)
#   diagnostic_file =  (Default)
#   refresh = 100 (Default)
```

Stan's behavior is fully specified by these configuration parameters. 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 contiues with a CSV header naming the columns of the output. For the default NUTS sampler in Stan 2.1.0, these are

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

The values for `lp__` indicate the log probability (up to an additive constant). The column headed by `accept_stat__` provides the Metropolis/slice acceptance statistic for each iteration.[8] 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 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.

[8] Acceptance 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 2011).

```
# 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 2013) 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 2013) 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
./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
```

```
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.*

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. See Section 9 for more information.

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.[9] 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

[9] Aligning columns when printing rows of varying scales presents a challenge. For each column, the program

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, Rubin, Dunson, and Vehtari 2013) and (Stan Development Team 2013) 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 2013) 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 $p(\theta|y) = \mathsf{Beta}(3, 9)$. 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

 method = optimize
   optimize
     algorithm = bfgs (Default)
```

calculates the 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.

```
      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 = 9.9999999999999995e-07 (Default)
         error = 9.9999999999999995e-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 `-3.81445e-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.

3. Models

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.

3.1. Example: hierarchical model, with inference

(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 similiar 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.

3.2. 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

[10]The 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 2013) 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.

3.3. 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.

3.4. 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 `lambda` 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. Furthermore, different

samplers and optimizers use different instantiations of the log probability function depending on the form and order of the derivative information they require; see Section 8 for details.

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 probabilty), 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 probabilty 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 probabilty function.[11] The gradients of the log probabilty 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.

3.5. 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 a special not-a-number value.

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 Jacobian 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

[11] For 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.

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

The transorm is $f(u) = \log u$, the inverse transform is $f^{-1}(v) = \exp v$, so the absolute log Jacobian 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.

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.

3.6. Model block

The model block defines 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.

3.7. 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 predictive posterior 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. Applying a hierarchical model then considering posterior inference appropriately adjusts for mulitple comparisons; see (Gelman, Hill, and Yajima 2012; Efron 2010) for discussion.

As a second illustration, the generated quantities block n 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.

3.8. 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 $[-2, 2]$ by default or another interval by specification. 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.

3.9. Variable definition and block execution summary

A table summarizing the point at which variables are read, written, or defined is provided in Figure 4. This table is defined assuming HMC or NUTS samplers, which require a log probability and gradient calculation for one or more leapfrog steps per iteration; see (Hoffman and Gelman 2011). Gelman and Hill (2007, p. 366) provide a taxonomy of the kinds of variables used in Bayesian models. Figure 5 contains Gelman and Hill's taxonomy aligned with the corresponding locations of declarations and definitions in Stan. Unmodeled data variables includes size constants and regression predictors. Modeled data variables include known outcomes or measurements. A data variable or constant literal is modeled in a Stan program if it (or a variable that depends on it) occurs on the left-hand side of a sampling statement. Unmodeled parameters are provided as data, and are either known or fixed to some value for convenience, a typical use being the parameters of a weakly informative prior for a parameter.

A modeled parameter is given a distribution by the model (usually dependent on data and correlated with other parameters). This can be done either by placing it (or a variable that depends on it) on the left-hand side of a sampling statement or by declaring the variable with a constraint (see Section 3.10).

Any variable that occurs in a (transformed) data block can also be provided instead as a

Block	Statements?	Action	Evaluated
`user initialization`	n/a	transform	chain
`random initialization`	n/a	randomize	chain
`data`	no	read	chain
`transformed data`	yes	evaluate	chain
`parameters`	no	inv. transform, Jacobian	leapfrog
		inv. transform, write	sample
`transformed parameters`	yes	evaluate	leapfrog
		write	sample
`model`	yes	evaluate	leapfrog
`generated quantities`	yes	evaluate	sample
		write	sample

Figure 4: *Each Stan program block admits certain actions that are evaluated at specific times during sampling. For example, the parameter initialization in the* `data` *block requires a read operation once per chain.*

Variable Categorization	Declaration Block
unmodeled data	`data`, `transformed data`
modeled data	`data`, `transformed data`
missing data	`parameters`, `transformed parameters`
modeled parameters	`parameters`, `transformed parameters`
unmodeled parameters	`data`, `transformed data`
generated quantities	`transformed data`, `transformed parameters`, `generated quantities`
loop indices	any loop statement
local variables	any statement block

Figure 5: *Variables of each categorization must be declared in specific blocks. Data may also be expressed using numeric literals.*

constant. So a user may decide to write `y ~ normal(0,1)` or to declare `mu` and `sigma` as data, and write `y ~ normal(mu,sigma)`. The latter choice allows the parameters to be changed without recompiling the model, but requires them to be specified as part of the input data.

Missing data is a new variable type included here. In order to perform inference on missing data, it must be declared as a parameter and modeled; see (Gelman *et al.* 2013) for a discussion of statistical models of missing data. Unlike BUGS, data may not contain a mixture of missing (i.e., `NA`) values and numeric values. Stan uses the built-in C++ floating-point (and integer) arithmetic, for which there is no equivalent of `NA`. See (Stan Development Team 2013) for strategies for coding missing data problems in Stan.

Generated quantities include simple transforms of data required for printing. For example, the variable of interest might be a standard deviation, resulting from transforming a precision parameter τ to a scale parameter $\sigma = \tau^{-1/2}$. A non-linearly transformed variable will have different effective sample sizes and $\hat{R}$ statistics than the variable it is derived from, so it is convenient to define variables of interest in the generated quantities block to calculate these statistics automatically. As seen in the example in Figure 3, generated quantities may also be used for event probability estimates.

The generated quantities block may also be used for forward simulations, generating values to make predictions or to perform posterior predictive checks; see (Gelman *et al.* 2013) for more information. The generated quantities block is the only location in Stan in which random-number generators may be applied explicitly (they are implicit in parameters).

Calculations in the generated quantities block do not impact the estimates of the parameters. In this way, generated quantities can be used for predictions with a behavior not unlike that of cut in BUGS (Lunn *et al.* 2012). Nevertheless, we recommend full Bayesian inference in general, and only use the generated quantities block for efficiency and clarity of code. For example, in many models, generating predictions for unseen data or replicating data for posterior predictive checks does not affect the estimation of the parameters of interest.

The list in Figure 5 is completed with two types of local variables, loop indices and traditional local variables. Unlike BUGS, Stan allows local variables to be declared and assigned. For example, it is possible to compute the sum of the squares of entries in an array or vector `y` as follows.[12]

```
{
  real sum_of_squares;
  sum <- 0;
  for (n in 1:N)
    sum <- sum + y[n] * y[n];
}
```

Local variables may not be declared with constraints, because there is no location at which it makes sense to test that they satisfy the constraints.

3.10. Implicit uniform priors

The default distribution for a variable is uniform over its support. For instance, a variable declared with a lower bound of 0 and an upper bound of 1 implicitly receives a Uniform$(0, 1)$

[12]The built-in squared norm function is the right way to implement sums of squares in Stan.

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.[13]

3.11. The non-centered parameterization for hierarchical models

Consider the following hierarchical logistic regression model fragment.

```
data {
  int<lower=1,upper=K> group[N];  // group of data item n
}
parameters {
  ...
  vector mu;              // mean coeff
  real<lower=0> sigma;    // coeff scale
  vector[K] beta;         // coeff for group k
}
model {
  beta ~ normal(mu,sigma);
  for (n in 1:N)
    y[n] ~ bernoulli_logit(beta[group[n]] * x[n]);
  ...
}
```

With this parameterization, the `beta[k]` are centered around `mu`. As a result, a change in `sigma` is amplified by the lower-level parameters and introduces a large change in density across the posterior. Unfortunately the expected density variation of an HMC transition is limited[14] and many transitions are required to explore the full posterior. The resulting sampler devolves into a random walk with high autocorrelations. Betancourt and Girolami (2013) provide a detailed analysis of the benefits of centered vs. non-centered parameterizations of hierarchical models in Euclidean and Riemannian HMC, with the conclusion that non-centered are best when data are sparse and centered when the data strongly identifies coefficients.

The following reparameterization employs a non-centered parameterization for comparison.

```
parameters {
  vector[K] beta_raw;
}
transformed parameters {
  vector[K] beta;
  beta <- mu + sigma * beta_raw;
```

[13] Improper posteriors are diagnosed automatically when parameters overflow to infinity during simulation.

[14] The limited density variation is ultimately a consequence of a constant metric. An additional benefit of RHMC are transitions that can cover much larger variations in density, making it uniquely suited to these models; see (Neal 2011).

```
}
model {
  beta ~ normal(0,1);
}
```

This reparameterization removes the particular correlations amongst the hierarchical variables that would otherwise limit the effectiveness of the samplers. We recommend starting with the more natural centered parameterization and moving to the non-centered parameterization if sampling of the coefficients does not mix well.[15]

4. 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.

4.1. 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*.

4.2. 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.

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

4.3. 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]`.[16]

[15] As the saying goes in computer science, it is easier to optimize a correct program than debug an optimized program.

[16] Arrays are stored in row-major order and matrices in column-major order.

Unlike integers, which may be used where real values are required, arrays of integers may not be used where real arrays are required.[17]

4.4. 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]` and `corr_matrix[K]`.

5. Expressions and type inference

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.

5.1. Expressions

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 each variable is declared statically and cannot change.

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`.

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

6. Statements

6.1. 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 statment 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 proportion, 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.

6.2. 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.

6.3. 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.

6.4. 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.[18] Stan provides the usual comparison operators and boolean operators to help define conditionals and condition-controlled while loops.

6.5. 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.

7. 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 be vectorized so that shared computations can be reused.

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

7.1. 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.[19] The full set of operators is listed in Figure 6.

7.2. 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 Figures 9 and 10.

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.

7.3. Matrix and linear algebra functions

Following the usual convention in mathematics, matrix and array indexing uses the usual square brackets (`[ ]`) operator, and begins indexing from 1. For example, if `Sigma` is a matrix, then `Sigma[m,n]` is the entry at row `m` and column `n`. Stan also allows slices of data structures to be returned, so that `Sigma[m]` is row `m` of the matrix `Sigma`.

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 `mu` are vectors and `Sigma` 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)' * 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, multiply by self transposed, e.g., the previous example could be written as `quad_form(Sigma, y - mu)`.

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

The full set of matrix and linear-algebra functions is listed in Figure 11; operators, which also apply to matrices and vectors, are listed in Figure 6.

7.4. 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 may be configured to throw exceptions or return $-\infty$ or a special not-a-number value (`NaN`) 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.

The list of probability functions is provided in Figures 12, 13, and 14.

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.[20] 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)`.

Vectorization

All of the univariate[21] probability functions in Stan are vectorized 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 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.

[20] Both vectorization 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.

[21] We are in the process of vectorizing the multivariate probability functions, but they are not all available in Stan 2.0. We are also in the process of vectorizing all of the special functions.

The advantage of using vectorization 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. In the example above, `y ~ normal(X * beta,sigma)`, the logarithm of `sigma` need only be computed once; if `y` is an N-vector, it also reduces the number of virtual function calls from N to 1.

8. 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.

8.1. Markov chain Monte Carlo samplers

Hamiltonian Monte Carlo

The MCMC samplers provided include Euclidean manifold Hamiltonian Monte Carlo (EHMC, or just HMC) (Duane, Kennedy, Pendleton, and Roweth 1987; Neal 1994, 2011) and the No-U-Turn sampler (NUTS) (Hoffman and Gelman 2011). 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 coherent motion through the posterior that dramatically reduces the autocorrelation of the resulting transitions.

Stan will soon have an implementation of Riemannian manifold Hamiltonian Monte Carlo (RHMC) (Girolami and Calderhead 2011), using the SoftAbs metric to robustly incorporate posterior curvature into the Hamiltonian simulation Betancourt (2012). Stan's MCMC framework also makes it straightforward to incorporate other metrics. This implementation will also generalize NUTS to Riemannian manifolds (Betancourt 2013). Both support adapting step sizes during warmup. RHMC with SoftAbs uses first, second, and third order derivatives to adapt to the local curvature of the posterior.

Metropolis-Hastings

Random-walk Metropolis-Hastings samplers with multivariate normal proposals have been prototyped, but not yet integrated into the released versions of Stan. The primary purpose of implementing Metropolis-Hastings is to provide baselines against which to measure more efficient approaches to sampling. The covariance matrix used can be adapted during warmup, assuming either equal diagonal covariance, diagonal covariance, or a full covariance matrix. Metropolis-Hastings does not require any derivatives of the log probability function, but the lack of gradient information induces random walk behavior, leading to slow mixing.

Ensemble samplers

Two flavors of ensemble samplers are also in the works: an affine invariant ensemble sampler

(Goodman and Weare 2010) and a differential evolution sampler (Ter Braak 2006). Ensemble samplers do not require derivative information, but typically require at least as many chains to be run as the number of dimensions in order to have full rank search of the posterior. They also require slightly different management of output due to a single sampler instance producing multiple correlated chains of draws.

8.2. 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 and marginal inference using approximate integration. All these algorithms require optimization steps.

BFGS

The default optimizer in Stan is the Broyden-Fletcher-Goldfarb-Shannon-Boyd (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.

9. Random number generation

Random number generation for Stan is done on a per-chain basis; ensemble samplers form a single joint chain over the ensemble. By specifying the chain being used, the random number generator can be skipped ahead sufficiently to avoid replication of subsequences of random numbers across chains.

The generated model code and underlying C++ algorithms provide template parameters for a class implementing the **Boost** random number generator concept.

By default, Stan uses linear congruential generators (L'Ecuyer 1988). This generator supports efficient skip-ahead.

9.1. Replicability

The Stan interfaces all allow random-number generator seeds to be specified explicitly. Execution uses a single base random number generator instance. Therefore, by specifying a seed,

Stan's behavior is deterministic. This is very useful for debugging purposes. Seeds can be generated randomly based on properties of the system time, but when they are, the seed used is printed as part of the output to allow it to be fully replicated.

10. Library dependencies

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

10.1. 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.

10.2. 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 flexiblity with respect to input types. Unfortunately, many of the expression templates that Eigen uses for efficient static anaysis 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.

11. Developer process

11.1. 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.

11.2. 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.

11.3. 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 MCMC standard error.

11.4. 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 2013) 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.

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 Univesity and Harvard University, who served as trial subjects for both Stan and (Gelman *et al.* 2013).

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

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

Figure 6: *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.*

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).

Finally, we would like to thank all 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.

And a special thanks to those who have contributed code for new features, Jeffrey Arnold, Yuanjun Gao, and Marco Inacio.

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

Figure 7: *Stan implements a variety of useful constants.*

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

Figure 8: *Stan implements both circular and hyperbolic trigonometric functions, as well as their inverses.*

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

Figure 9: *Stan implements many special and transcendental functions.*

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

Figure 10: *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_post_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 11: *A large suite of matrix functions admits efficient multivariate model implementation in Stan.*

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

Figure 12: *Most common probability distributions have explicitly implemented in Stan.*

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

Figure 13: *Probability functions (continued)*

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

Figure 14: *Probability functions (continued 2)*

References

Bell B, Burke J (2008). "Algorithmic differentiation of implicit functions and optimal values." In *Advances in Automatic Differentiation*. Springer-Verlag.

Betancourt M (2012). "A General Metric for Riemannian Manifold Hamiltonian Monte Carlo." *arXiv*, **1212**(4693). URL http://arxiv.org/abs/1212.4693.

Betancourt M (2013). "Generalizing the No-U-Turn Sampler to Riemannian Manifolds." *arXiv*, **1304**(1920). URL http://arxiv.org/abs/1304.1920.

Betancourt M, Girolami M (2013). "Hamiltonian Monte Carlo for Hierarchical Models." *arXiv*, **1312**(0906). URL http://arxiv.org/abs/1312.0906.

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.

Gay D (2005). "Semiautomatic Differentiation for Efficient Gradient Computations." In HM Bücker, GF Corliss, P Hovland, U Naumann, B Norris (eds.), *Automatic Differentiation: Applications, Theory, and Implementations*, volume 50 of *Lecture Notes in Computational Science and Engineering*. Springer, New York.

Gelman A, Carlin JB, Stern HS, Rubin DB, Dunson D, Vehtari A (2013). *Bayesian Data Analysis*. 3rd edition. CRC Press, London.

Gelman A, Hill J (2007). *Data Analysis Using Regression and Multilevel-Hierarchical Models*. Cambridge University Press, Cambridge, United Kingdom.

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.

Giles MB (2008). "An extended collection of matrix derivative results for forward and reverse mode algorithmic differentiation." *Technical Report NA-08/01*, Oxford University Computing Laboratory.

Girolami M, Calderhead B (2011). "Riemann manifold Langevin and Hamiltonian Monte Carlo methods." *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, **73**(2), 123–214.

Goodman J, Weare J (2010). "Ensemble samplers with affine invariance." *Communications in Applied Mathematics and Computational Science*, **5**(1), 65–80.

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 (2011). "The No-U-Turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo." *arXiv*, **1111**(4246). URL http://arxiv.org/abs/1111.4246.

L'Ecuyer P (1988). "Efficient and Portable Combined Random Number Generators." *Communications of the ACM*, **31**, 742–749, 774.

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/k2)." *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/.

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 (2013). *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 (2013). *Stan Modeling Language User's Guide and Reference Manual*. Version 2.1.0, URL http://mc-stan.org/manual.html.

Ter Braak CJF (2006). "A Markov Chain Monte Carlo version of the genetic algorithm Differential Evolution: easy Bayesian computing for real parameter spaces." *Statistical Computing*, **16**, 239–249.

van Heesch D (2011). "Doxygen: Generate Documentation from Source Code." http://www.stack.nl/~dimitri/doxygen/index.html.

Vandevoorde D, Josuttis NM (2002). *C++ Templates: The Complete Guide*. Addision-Wesley. ISBN 978-0201734843.

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 http://www.jstatsoft.org/
published by the American Statistical Association http://www.amstat.org/
Volume VV, Issue II *Submitted:* yyyy-mm-dd
MMMMMM YYYY *Accepted:* yyyy-mm-dd