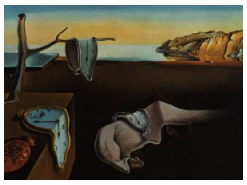

Journal of Statistical Software

January 2017, Volume 76, Issue 1. *doi: 10.18637/jss.v076.i01*

Stan: A Probabilistic Programming Language

Bob Carpenter
Columbia University

Andrew Gelman
Columbia University

Matthew D. Hoffman
Adobe Creative Technologies Lab

Daniel Lee
Columbia University

Ben Goodrich
Columbia University

Michael Betancourt
Columbia University

Marcus A. Brubaker
York University

Jiqiang Guo
NPD Group

Peter Li
Columbia University

Allen Riddell
Indiana University

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 2.14.0, Stan provides full Bayesian inference for continuous-variable models through Markov chain Monte Carlo methods such as the No-U-Turn sampler, an adaptive form of Hamiltonian Monte Carlo sampling. Penalized maximum likelihood estimates are calculated using optimization methods such as the limited memory Broyden-Fletcher-Goldfarb-Shanno algorithm.

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

Stan can be called from the command line using the **cmdstan** package, through R using the **rstan** package, and through Python using the **pystan** package. All three interfaces support sampling and optimization-based inference with diagnostics and posterior analysis. **rstan** and **pystan** also provide access to log probabilities, gradients, Hessians, parameter transforms, and specialized plotting.

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

1. Introduction

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

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

Stan has the interfaces **cmdstan** for the command line shell, **pystan** for Python (Van Rossum *et al.* 2016), and **rstan** for R (R Core Team 2016). Stan also provides packages wrapping **cmdstan**, including **MatlabStan** for MATLAB, **Stan.jl** for Julia, **StataStan** for Stata, and **MathematicaStan** for Mathematica. These interfaces run on Windows, Mac OS X, and Linux, and are open-source licensed.

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

2. Core functionality

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

2.1. Program 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. 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:

```
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`.

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

```
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: Stan program to estimate chance of success in the independent Bernoulli model.

2.2. Data format

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

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

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

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

2.3. Compiling the model

After a C++ compiler and **make** are installed,[2] 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.[3]

```
> cd $stan
> make replication/bernoulli
```

[2]Appropriate versions are built into Linux. The **RTools** package suffices for Windows; it is available from `https://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.

[3]Before the first model is built, **make** must build the model translator (target `bin/stanc`) and posterior summary tool (target `bin/stansummary`), 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.

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

2.4. Running the sampler

Command to sample from the model

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

```
> cd $stan/replication/bernoulli
> ./bernoulli sample data file=bernoulli.data.R \
  random seed=2261934443 id=1 output file=output1.csv
```

This command specifies that sampling should be performed with the model instantiated using the data in the specified file. The backslash (\) indicates a continued input line. For Windows, the `./` before the command should be removed and the line continuation replaced with a caret (^).

The aruments on the continued line are optional and included for the sake of replicability. Bit-by-bit replicability requires fixing the operating system version, central processing unit, C++ compiler, and the compiler settings.

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 sample 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 = 1
data
  file = bernoulli.data.R
init = 2 (Default)
random
  seed = 2261934443
output
  file = output1.csv
  diagnostic_file =  (Default)
  refresh = 100 (Default)

Gradient evaluation took 8e-06 seconds
1000 transitions using 10 leapfrog steps per transition would take 0.08 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: 0.01042 seconds (Warm-up)
               0.019595 seconds (Sampling)
               0.030015 seconds (Total)
```

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

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

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

Help

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

```
> ./bernoulli help-all
```

The sampler and its configuration are described at greater length in the manual (Stan Development Team 2016).

Sample file output

The output CSV file (comma-separated values), written explicitly to `output1.csv`, starts with a summary of the configuration parameters for the run.

```
# stan_version_major = 2
# stan_version_minor = 14
# 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 = 1
# data
#   file = bernoulli.data.R
# init = 2 (Default)
# random
#   seed = 2261934443
# output
#   file = output1.csv
#   diagnostic_file =  (Default)
#   refresh = 100 (Default)
...
```

Stan's behavior is fully specified by these configuration parameters, almost all of which have default values. 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 explicitly specified with value 2261934443) and the chain identifier (here explicitly specified as 1). The identifier is used to advance the underlying pseudorandom number generator a sufficient number of values that using multiple chains with the same seed and different identifiers will draw from different subsequences of the pseudorandom number stream determined by the seed.

The output continues with a CSV header naming the columns of the output. For the default NUTS sampler in Stan 2.14.0, the output is as follows (on one line without the backslash).

```
lp__,accept_stat__,stepsize__,treedepth__,n_leapfrog__,\
divergent__,energy__,theta
```

The label `lp__` is for log densities (up to an additive constant), `accept_stat__` is for acceptance probabilities,[4] `stepsize__` is for the leapfrog integrator's step size for simulating the Hamiltonian, `treedepth__` is the depth of tree explored by the no-U-turn sampler (log base 2 of the number of log density and gradient evaluations), `n_leapfrog__` is the number of density and gradient evaluations, `divergent__` is a flag indicating a numerical instability during numerical integration resulting in the Hamiltonian not being conserved, and `energy__` is the Hamiltonian value. The rest of the header will be the names of parameters; in this example, `theta` is the only parameter.

The results of step size and mass matrix adaptation are printed as comments.

```
# Adaptation terminated
# Step size = 1.66784
# Diagonal elements of inverse mass matrix:
# 0.465594
```

Unless adaptation is turned off, Stan uses the first half of the iterations to estimate a mass matrix and step size for numerical integration of the the Hamiltonian system. Stan uses a diagonal mass matrix by default, but may also be configured to use a dense mass matrix or unit mass matrix. The inverse mass matrix is estimated by regularizing the sample (co)variance of the latter half of the warmup iterations; see (Stan Development Team 2016) for full details.

The rest of the file contains the sample, one draw per line, matching the header; here the parameter `theta` is the final value printed on each line, and each line corresponds to a draw from the posterior. The warmup iterations are excluded by default, but may be included with appropriate command configuration. The file ends with comments reporting the elapsed time.

```
-6.74818,0.60304,1.66784,1,1,0,7.27164,0.247792
-7.35902,0.800678,1.66784,1,1,0,7.44313,0.401688
```

[4]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 the trajectory states in the proposed tree; the original NUTS algorithm used a slice sampling algorithm for rejection (Neal 2003; Hoffman and Gelman 2014) whereas Stan 2.14.0 uses a multinomial sampler with probabilities given by the Hamiltonian (Betancourt 2016).

```
-8.12629,0.412962,1.66784,2,3,0,8.97432,0.483505
...
-8.41114,1,1.66784,1,1,0,9.79546,0.0770631
-8.09041,1,1.66784,1,1,0,8.85713,0.0892462
-6.76262,1,1.66784,1,1,0,7.38722,0.22906
#
#  Elapsed Time: 0.01042 seconds (Warm-up)
#                0.019595 seconds (Sampling)
#                0.030015 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. In this example, the Jacobian is the absolute derivative of the inverse logit function; see (Stan Development Team 2016) for the constrained parameter 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 (Gelman and Rubin 1992; Gelman, Carlin, Stern, Dunson, Vehtari, and Rubin 2013). Three more chains of draws can be created as follows.

```
./bernoulli sample data file=bernoulli.data.R random seed=2261934443 \
            id=1 output file=output1.csv
./bernoulli sample data file=bernoulli.data.R random seed=2261934443 \
            id=2 output file=output2.csv
./bernoulli sample data file=bernoulli.data.R random seed=2261934443 \
            id=3 output file=output3.csv
```

These calls illustrate how additional parameters are specified directly on the command line following the hierarchy given in the output. The backslash (\) at the end of a line indicates that the command continues on the next 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. Although the same seed is used for each chain, the random numbers will in fact be independent as the chain identifier is used to skip the pseudorandom number generator ahead. Stan supplies a command line program `bin/stansummary` to summarize the output of one or more MCMC chains. Given a directory containing output from sampling,

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

```
output1.csv         output2.csv         output3.csv         output4.csv
```

posterior summaries are printed using

```
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 (0.010, 0.011, 0.0100, 0.0099) seconds, 0.041 seconds total
Sampling took (0.020, 0.021, 0.021, 0.020) seconds, 0.082 seconds total

                Mean     MCSE  StdDev     5%    50%   95%  N_Eff  N_Eff/s    R_hat
lp__            -7.3  1.9e-02    0.78   -8.8   -7.0  -6.8   1753    21384  1.0e+00
accept_stat__   0.86  3.7e-02    0.18   0.50   0.94   1.0     24      288  1.0e+00
stepsize__       1.5  1.5e-01    0.21    1.3    1.7   1.8    2.0       24  6.9e+13
treedepth__      1.3  1.3e-01    0.46    1.0    1.0   2.0     13      163  1.1e+00
n_leapfrog__     1.9  1.6e-01    1.00    1.0    1.0   3.0     37      448  1.0e+00
divergent__     0.00  0.0e+00    0.00   0.00   0.00  0.00   4000    48801      nan
energy__         7.8  2.6e-02     1.1    6.8    7.4   9.9   1652    20158  1.0e+00
theta           0.25  2.8e-03    0.12  0.082   0.24  0.47   1851    22586  1.0e+00

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

Figure 2: Output summary for the Bernoulli estimation model in Figure 1.

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

The output is shown in Figure 2.[5] Each row of the output summarizes a different value whose name is provided in the first column. These correspond to the columns in the output CSV files. The analysis includes estimates of the posterior mean (`Mean`) and standard deviation (`StdDev`). Quantiles for the median (`50%`) and the 90% posterior interval (`5%`, `95%`) are also displayed; the quantiles printed can be configured.

The remaining columns in the output provide an analysis of the sampling and its efficiency. The convergence diagnostic that is built into the `bin/stansummary` 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 *et al.* 2013) and (Stan Development Team 2016) for definitions.

The column `N_eff` reports the effective sample size for a chain. Because MCMC methods produce a sample containing correlated draws in each chain, estimates such as posterior means are not expected to be as precise as they would be with truly independent draws. The

[5]Aligning columns when printing rows of varying scales presents a challenge. For each column, the program 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 `MCSE` (Markov chain standard error) column.

effective sample size is an estimate of the number of independent draws that would lead to the same expected precision. 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 effective sample size (`sd / sqrt(n_eff)`). Geyer (2011) provides a thorough introduction to effective sample size and MCSE estimation. Stan uses more conservative estimates based on both within-chain and cross-chain convergence; see (Gelman *et al.* 2013) and (Stan Development Team 2016) for motivation and definitions.

Because estimation accuracy is governed by the square root of the effective sample size, effective sample size per second (or its inverse) 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 to generate a target effective sample size.

In this example, the estimated effective sample size (`n_eff`) is 1776, which is far greater than we typically need for inference. The posterior mean here is estimated to be 0.25 with an MCSE of 0.003. Because the model is conjugate, the exact posterior is known to be $p(\theta \mid y) = \mathsf{Beta}(\theta \mid 3, 9)$, which has a mean of $3/(3 + 9) = 0.25$ and a mode of $(3 - 1)/(3 + 9 - 2) = 0.2$.

2.6. Estimators

Stan provides several ways to compute point estimates of parameters. The standard Bayesian approach is to use posterior means or medians, as computed by MCMC. The posterior mode, when it exists, is another popular estimator (providing what is sometimes called the *maximum a posteriori* (MAP) estimate). Stan programs can be interpreted as defining penalized log likelihood functions rather than posterior log densities, in which case the mode is the penalized maximum likelihood estimate (MLE).

Modes with optimization

The posterior mode (or penalized MLE) of the parameters conditioned on the data given the model can be found by using one of Stan's built-in optimizers.The following command invokes optimization for the Bernoulli model using default configuration parameters for everything but the random seed, which is included for replicability following the line-continuation backslash.

```
> ./bernoulli optimize data file=bernoulli.data.R \
  random seed=2261934443 output file=opt-fit.csv

method = optimize
  optimize
    algorithm = lbfgs (Default)
      lbfgs
        init_alpha = 0.001 (Default)
        tol_obj = 9.9999999999999998e-13 (Default)
        tol_rel_obj = 10000 (Default)
        tol_grad = 1e-08 (Default)
        tol_rel_grad = 10000000 (Default)
        tol_param = 1e-08 (Default)
        history_size = 5 (Default)
    iter = 2000 (Default)
    save_iterations = 0 (Default)
id = 0 (Default)
data
  file = bernoulli.data.R
init = 2 (Default)
random
  seed = 2261934443
output
  file = output.csv (Default)
  diagnostic_file =  (Default)
  refresh = 100 (Default)

initial log joint probability = -10.8352
    Iter      log prob        ||dx||      ||grad||
       6      -5.00402   0.000165244   5.44531e-07

       alpha      alpha0  # evals  Notes
           1           1        9

Optimization terminated normally:
  Convergence detected: relative gradient magnitude is below tolerance
```

The final lines of the output indicate normal termination after seven iterations by convergence of the objective function (here the log density or penalized log likelihood) to within the default tolerance of `1e-08`. The other values include final value of the log probability function (`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 density, gradient, or parameter values are within their specified tolerance. The default optimizer uses the limited memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS) algorithm, a quasi-Newton method which employs gradients and a memory and time efficient approximation to the Hessian (Nocedal and Wright 2006).

Optimizer output file

By default, optimization results are written into a valid CSV file, here specified to be `opt-fit.csv`.

```
# stan_version_major = 2
# stan_version_minor = 14
# stan_version_patch = 0
# model = bernoulli_model
# method = optimize
#   optimize
#     algorithm = lbfgs (Default)
#       lbfgs
#         init_alpha = 0.001 (Default)
#         tol_obj = 9.9999999999999998e-13 (Default)
#         tol_rel_obj = 10000 (Default)
#         tol_grad = 1e-08 (Default)
#         tol_rel_grad = 10000000 (Default)
#         tol_param = 1e-08 (Default)
#         history_size = 5 (Default)
#     iter = 2000 (Default)
#     save_iterations = 0 (Default)
# id = 0 (Default)
# data
#   file = bernoulli.data.R
# init = 2 (Default)
# random
#   seed = 2261934443
# output
#   file = output.csv (Default)
#   diagnostic_file =  (Default)
#   refresh = 100 (Default)
lp__,theta
-5.00402,0.2
```

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 density, `lp__`, and the single parameter name, `theta`. The next line shows that the posterior mode for `theta` is 0.200002, 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 density. This ensures modes are defined with respect to the constrained parameter space as declared in the parameters block and used in the model specification.

2.7. Diagnostic mode

Stan provides a diagnostic mode that evaluates the log density and its gradient at the initial parameter values (either user supplied or generated randomly based on the specified or default seed). The seed and chain ID are set so that the point evaluated is the initialization of the first MCMC chain run above.

```
> ./bernoulli diagnose data file=bernoulli.data.R \
   id=1 random seed=2261934443

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 = 2261934443
output
  file = output.csv (Default)
  diagnostic_file =  (Default)
  refresh = 100 (Default)


TEST GRADIENT MODE

 Log probability=-12.4362

 param idx           value           model     finite diff           error
         0        0.943403        -5.63744        -5.63744     1.06921e-09
```

Here, a random initialization is used and the initial log density is `-12.4362` and the single parameter `theta`, here represented by index 0, has a value of `0.943403` on the unconstrained scale (inverse-logit transformed to 0.7198 on the constrained scale). The derivative supplied by the model and by a finite differences calculation are the same to within `1.06921e-09`. Non-finite log densities or derivatives indicate a problem with the model in terms of constraints on parameter values, function input constraints being violated, boundary conditions arising in function evaluations, and sometimes overflow or underflow issues with floating-point calculations. Large relative discrepancies between the model's gradient calculation and finite differences can indicate a bug in the model or even in Stan's algorithmic differentiation for a function in the model.

2.8. Roadmap for the rest of the paper

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

3. Data types

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

3.1. Primitive types

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

3.2. Vector and matrix types

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

Multiple indexing may be applied, with syntax following that of R and MATLAB. For example, if `a` is a vector containing at least four elements, then `a[2:4]` has elements `a[2]`, `a[3]`, and `a[4]`. Either or both bounds may be omitted, in which case they default to the first and last element in the vector. For example, `a[3:]` has two fewer elements than `a` and is defined so that `a[3:][i]` evaluates to `a[2 + i]`. If `n` is an array of integers, then `a[n]` is a vector with the same size as `n` and is defined so that `a[n][i]` evaluates to `a[n[i]]`.

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

3.3. Array types

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

Higher-dimensional arrays are intrinsically arrays of arrays. An entry in a two-dimensional array `y` may be accessed as `y[1,2]`. The expression `y[1]` by itself denotes the one-dimensional array whose values correspond to the first row of `y`. Thus `y[1][2]` has the same value as `y[1,2]`.[6] Unlike integers, which may be used where real values are required, arrays of integers may not be used where real arrays are required.[7]

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

3.4. Constrained variable types

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

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

Vector types may be constrained to be unit simplexes (all entries non-negative and summing to one), unit length vectors (sum of squares is one), ordered (entries are in ascending order),

[6] Arrays are stored internally in row-major order and matrices in column-major order. Stan's input and output matches R's use of column-major order, with arrays being converted internally.

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

positive ordered (entries in ascending order, all non-negative), using the types `simplex[K]`, `unit_vector[K]`, `ordered[K]`, and `positive_ordered[K]`, where `K` is the size of the vector.

Matrices may be constrained to be covariance or precision matrices (symmetric, positive definite) or correlation matrices (symmetric, positive definite, unit diagonal), using the types `cov_matrix[K]` and `corr_matrix[K]`. For efficient and stable arithmetic, matrices may also be defined to be Cholesky factors of covariance or correlation matrices, using the types `cholesky_factor_cov[K]` and `cholesky_factor_corr[K]`.

3.5. Expressions

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

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

Type inference

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

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

Type promotion

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

4. Top-level blocks and program execution

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

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

4.1. Data block

A Stan program starts with an (optional) data block, which declares the data required to fit

```
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 = lambda * kappa;        // prior success count
  real<lower=0> beta = (1 - lambda) * kappa;  // prior failure count
}
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 = mean(theta);    // 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
  for (j in 1:J) {
    above_avg[j] = (theta[j] > avg);
    rnk[j] = rank(theta, j) + 1;
    highest[j] = (rnk[j] == 1);
  }
}
```

Figure 3: Hierarchical binomial model with posterior predictive quantities, coded in Stan.

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.[8] Missing data models may still be coded in Stan, but the missing values must be declared as parameters; see (Stan Development Team 2016) 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

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

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. The constraint language for Stan is not strong enough to restrict each y[j] to be less than or equal to n[j].

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

4.2. Transformed data block

The Stan program 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 constants needed by the program should 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 - mean(x)) / sd(x);
}
```

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

4.3. Parameter block

The parameter block in the program in Figure 3 defines three parameters. The parameter theta[j] represents the probability of success in group j. The prior on each theta[j] is parameterized by a mean chance of success lambda and count kappa. Both theta[j] and lambda are constrained to fall between zero and one. The Pareto distribution requires a strictly positive lower bound, so kappa is constrained to be greater than or equal to a conservative bound of 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 density is evaluated. This may be multiple times per iteration of a sampling or optimization algorithm.

Implicit change of variables to unconstrained space

The probability distribution defined by a Stan program is intended to have unconstrained support (i.e., no points of zero probability), which greatly simplifies the task of writing samplers or optimizers. To achieve unbounded support, variables declared with constrained support are transformed to an unconstrained space. For instance, variables declared on $[0, 1]$ are log-odds transformed and non-negative variables declared to fall in $[0, \infty)$ are log transformed.[9] More complex transforms are required for simplexes (a reverse stick-breaking transform) and covariance and correlation matrices (Cholesky factorization). The dimensionality of the resulting probability function may change as a result of the transform. For example, a $K \times K$ covariance matrix requires only $\binom{K}{2} + K$ unconstrained parameters, and a K-simplex requires only $K - 1$ unconstrained parameters.

The unconstrained parameters over which the model is defined are inverse transformed back to satisfy their constraints before executing any statements in the model block. To account for the change of variables, the log absolute Jacobian determinant of the inverse transform is added to the overall log density.[10] The gradients of the log density include the Jacobian term.

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

4.4. Transformed parameters block

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

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

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

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

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

[9] Values on the boundaries will be transformed to positive or negative infinity as is the standard for floating-point computer arithmetic.

[10] For optimization, the Jacobian adjustment is suppressed to guarantee the optimizer finds the maximum of the log density function on the constrained parameters. The calculation of the Jacobian is controlled by a template parameter in C++.

```
transformed parameters {
  real v = log(u);
}
model {
  v ~ normal(0, 1);  // distribution on transformed parameter
  target += u;       // log density Jacobian adjustment
}
```

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

Values of transformed parameters are saved in the output along with the parameters. As an alternative, local variables may be used to define temporary values in the model block.

4.5. Model block

The purpose of the model block is to define the log density 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, scalar values will be repeated as many times as necessary.

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

Implicit uniform priors

The default distribution for a parameter is uniform over its declared (constrained) support. For instance, a variable declared with a lower bound of 0 and an upper bound of 1 implicitly receives a $\textsf{Uniform}(0, 1)$ distribution. These implicit uniform priors are improper if the variable has unbounded support. For instance, the uniform distributions over real values with upper and lower bounds, simplexes, and correlation matrices is proper, but the uniform distribution over unconstrained or one-side constrained reals, ordered vectors or covariance matrices are not proper.

Stan does not require proper priors, but if the posterior is improper, Stan will halt with an error message.[11]

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

4.6. Generated quantities block

The (optional) generated quantities block allows values that depend on parameters and data, but do not affect estimation, to be defined efficiently. It may be used to calculate predictive inferences as well as to carry out forward simulation for posterior predictive checks; see (Gelman *et al.* 2013) for examples. Pseudorandom number generators are also available in the generated quantities block. The generated quantities block is called only once per iteration, not once per log density evaluation. Calculations in the generated quantities block are also more efficient because they do not require gradients.

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

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

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

4.7. Initialization

Stan's samplers and optimizers all start from either random or user-supplied values for each parameter. User-supplied initial values are validated and transformed to the underlying unconstrained space; if a parameter value does not satisfy its declared constraints, the program exits and an informative error message is printed. For each variable that is not initialized, the built-in pseudorandom number generator is called once per unconstrained variable dimension. The default initialization is to randomly generate values uniformly on $[-2, 2]$; another symmetric interval around zero may be configured. This supplies diffuse starting points when transformed back to the constrained scale, facilitating convergence diagnostics (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 locate the high mass region of the posterior.

Although Stan is quite effective at converging from diffuse random initializations, the user may supply their own initial values for sampling, optimization, or diagnosis. The top-level command line option configures a file from which to read initial values for parameters in the same R dump format used for data.

5. Statements

5.1. Assignment and sampling

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

Log density accumulator

There is an implicitly defined log density accumulator, the current value of which is available through the nullary function `target()`.

Sampling statements

A sampling statement is nothing more than shorthand for incrementing the log density accumulator. 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 incrementing the log density directly with the value of the log probability density function for the normal distribution using the target increment statement

```
target += normal_lpdf(beta | 0, 1);
```

Define variables before sampling statements

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

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

Direct definition of probability functions

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

```
target += -0.5 * beta^2;
```

If `beta` is of type `vector`, replace the square with the vector product `beta' * beta`, or the more efficient `dot_self(beta)`. Distributions whose probability functions are not built directly into Stan can be implemented directly in this fashion.

5.2. Sequences of statements and execution order

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

Blocks and variable scope

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

Variables declared in the top-level blocks (data, transformed data, parameters, transformed parameters, generated quantities), may only be assigned to in the block in which they are declared. They may be used at any point after they are declared, including subsequent blocks.

5.3. Whitespace, semicolons, and comments

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

Stan supports the line comment style of C++, using two forward slashes (//) to comment out the rest of a line; this is the one location where the content of whitespace matters. Stan also supports 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 comments 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. Camel case is valid Stan syntax, but dots may not be used in variable names.

5.4. Control structures

Stan supports the same kind of explicitly bounded for loops as found in BUGS and R. Like R, but unlike BUGS, Stan supports while loops and conditional (if-then-else) statements, as well as break and continue statements.[12] Stan provides the usual comparison operators and boolean operators to help define conditionals and condition-controlled while loops.

5.5. Print and reject statements

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

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

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

[12]BUGS omits these control structures because they would introduce data- or parameter-dependency into the directed, acyclic graph defined by model.

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

The print statement added at the last line will print the values in the matrix before the validation occurs at the end of the transformed parameters block.

Stan also supports reject statements which may be used to halt execution and return a meaningful error message. Like print statements, they take any number of string and expression arguments.

```
if (n > size(x))
  reject("Index out of bounds, n = ", n,
         "; required n < size(x) = ", size(x));
```

6. Function and distribution library

Stan is translated to C++ code that depends on the Stan math library to compute special functions, probability functions, matrix arithmetic and linear algebra, and solutions to ordinary differential equation (Carpenter, Hoffman, Brubaker, Lee, Li, and Betancourt 2015). In order to support the efficient algorithmic differentiation required to calculate gradients, Hessians, and higher-order derivatives in Stan, C++ functions must be templated separately for each argument. In order for these functions to be efficient in computing both values and derivatives, they need to operate directly on vectors of arguments so that shared computations can be reused. For example, if `y` is a vector and `sigma` is a scalar, the logarithm of `sigma` need only be evaluated once in order to compute the normal density for every member of `y` in

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

6.1. Basic operators

Stan supports all of the basic C++ arithmetic operators, boolean operators, and comparison operators. In addition, it extends the arithmetic operators to matrices and includes elementwise matrix operators, left and right matrix division, and transposition.[13]

6.2. Special functions

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

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.

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

This allows a more concise notation for generalized linear models as well as more efficient and arithmetically stable execution.

6.3. Matrix and linear algebra functions

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

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.

Matrix operators use the types of their operands to determine the type of the result. For example, 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, and multiply by self transposed; e.g., the previous example `(y - mu)' * Sigma * (y - mu)` could be coded as as `quad_form_diag(Sigma, y - mu)`.

6.4. Probability functions

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

All probability functions are defined on the log scale to avoid underflow. They are all named with the suffix `_lpdf` or `_lpmf` depending on whether they are density or mass functions, e.g., `normal_lpdf` is the log-scale normal distribution probability density function and `poisson_lpmf` is the log scale Poisson probability mass function.

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

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.[14] 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_lpdf(y | 0, sigma)`.

Vector arguments and shared computations

All of the univariate probability functions in Stan are implemented so that they accept arrays or vectors as arguments. For example, although the basic signature of the probability function `normal_lpdf(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.

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

7. Built-in inference engines

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

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

7.1. Markov chain Monte Carlo samplers

Hamiltonian Monte Carlo

The MCMC samplers provided include Euclidean Hamiltonian Monte Carlo (EHMC, which in much of the literature is referenced as simply HMC, Duane, Kennedy, Pendleton, and Roweth 1987; Neal 1994, 2011) and the no-U-turn sampler (NUTS, Hoffman and Gelman 2014; Betancourt 2016). 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 density function to generate coherent motion through the posterior that dramatically reduces the autocorrelation of the resulting transitions.

7.2. Optimizers

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

L-BFGS

The default optimizer in Stan is the limited-memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS) optimizer (Nocedal and Wright 2006). L-BFGS is a quasi-Newton optimizer that evaluates gradients directly, then uses a limited history of gradients to update an approximation to the Hessian.

Conjugate gradient

Stan provides a standard form of conjugate gradient optimization (Nocedal and Wright 2006). As its name implies, conjugate gradient optimization requires gradient evaluations.

Newton

Additionally, Stan implements a straightforward version of Newton's algorithm (Nocedal and Wright 2006), using gradients and Hessians.

8. Conclusion

Stan is a probabilistic programming language which allows users to specify a broad range of statistical models involving continuous parameters by coding their log posteriors (or penalized maximum likelihood) up to a proportion. Random variables are first-class objects and behave as expected under transformations. Stan provides full Bayesian inference for posterior expectations including parameter estimation and posterior predictive inference by defining appropriate derived quantities of interest. Stan implements full Bayesian inference with adaptive Hamiltonian Monte Carlo sampling and penalized maximum likelihood estimation with quasi-Newton optimization. Stan is implemented in standards-compliant C++, runs on all major computer platforms, and can be used interactively through interface languages including R and Python.

Appendix A describes the development process and Appendix B describes the library dependencies for Stan.

Computational details

The commands in this paper were run using **cmdstan** 2.14.0 on a Mac OS X version 10.10.5 on a Macbook Pro (Retina Mid 2012) with 2.3 GHz Intel Core i7 and **clang++** version Apple LLVM version 7.0.2 (clang-700.1.81) installed through **Xcode**. All examples were compiled at optimization level 3 and run in a fresh default terminal.

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 would like to particularly single out the students in Andrew Gelman's Bayesian data analysis courses at Columbia University and Harvard University, who served as trial subjects for both Stan and (Gelman *et al.* 2013).

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

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

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

Finally, we would like to thank the two anonymous referees.

References

Betancourt M (2016). "Identifying the Optimal Integration Time in Hamiltonian Monte Carlo." arXiv:1601.00225 [stat.ME]. URL `https://arxiv.org/abs/1601.00225`.

Carpenter B, Hoffman MD, Brubaker M, Lee D, Li P, Betancourt M (2015). "The Stan Math Library: Reverse-Mode Automatic Differentiation in C++." arXiv:1509.07164 [cs.MS]. URL `https://arxiv.org/abs/1509.07164`.

Chacon S (2009). *Pro **Git***. Apress. doi:10.1007/978-1-4302-1834-0.

Cohen SD, Hindmarsh AC (1996). "**CVODE**, A Stiff/Nonstiff ODE Solver in C." *Computers in Physics*, **10**(2), 138–143.

Driessen V (2010). "A Successful **Git** Branching Model." URL http://nvie.com/posts/a-successful-git-branching-model/.

Duane AD, Kennedy A, Pendleton B, Roweth D (1987). "Hybrid Monte Carlo." *Physics Letters B*, **195**(2), 216–222. doi:10.1016/0370-2693(87)91197-x.

Efron B (2010). *Large-Scale Inference: Empirical Bayes Methods for Estimation, Testing, and Prediction.* Institute of Mathematical Statistics Monographs. Cambridge University Press.

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

Gelman A, Hill J, Yajima M (2012). "Why We (Usually) Don't Have to Worry about Multiple Comparisons." *Journal of Research on Educational Effectiveness*, **5**, 189–211. doi:10.1080/19345747.2011.618213.

Gelman A, Rubin DB (1992). "Inference from Iterative Simulation Using Multiple Sequences." *Statistical Science*, **7**(4), 457–472. ISSN 0883-4237. doi:10.1214/ss/1177011136.

Geyer CJ (2011). "Introduction to Markov Chain Monte Carlo." In *Handbook of Markov Chain Monte Carlo.* Chapman & Hall/CRC.

Google (2016). "**googletest**: Google C++ Testing Framework." URL http://code.google.com/p/googletest/.

Guennebaud G, Jacob B, *et al.* (2010). "**Eigen**, Version 3." http://eigen.tuxfamily.org.

Hindmarsh AC, Brown PN, Grant KE, Lee SL, Serban R, Shumaker DE, Woodward CS (2005). "**SUNDIALS**: Suite of Nonlinear and Differential/Algebraic Equation Solvers." *ACM Transactions on Mathematical Software*, **31**(3), 363–396. doi:10.1145/1089014.1089020.

Hoffman MD, Gelman A (2014). "The No-U-Turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo." *Journal of Machine Learning Research*, **15**, 1593–1623.

Lunn D, Jackson C, Best N, Thomas A, Spiegelhalter DJ (2012). *The BUGS Book – A Practical Introduction to Bayesian Analysis.* Chapman & Hall/CRC.

Lunn D, Spiegelhalter DJ, Thomas A, Best N (2009). "The BUGS Project: Evolution, Critique, and Future Directions." *Statistics in Medicine*, **28**, 3049–3067. doi:10.1002/sim.3680.

Lunn D, Thomas A, Best NG, Spiegelhalter DJ (2000). "**WinBUGS** – A Bayesian Modelling Framework: Concepts, Structure, and Extensibility." *Statistics and Computing*, **10**, 325–337. doi:10.1023/a:1008929526011.

Metropolis N, Rosenbluth AW, Rosenbluth MN, Teller AH, Teller E (1953). "Equation of State Calculations by Fast Computing Machines." *Journal of Chemical Physics*, **21**, 1087–1092. doi:10.1063/1.1699114.

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 & Hall/CRC.

Neal RM (1994). "An Improved Acceptance Procedure for the Hybrid Monte Carlo Algorithm." *Journal of Computational Physics*, **111**, 194–203. doi:10.1006/jcph.1994.1054.

Neal RM (2003). "Slice Sampling." *The Annals of Statistics*, **31**(3), 705–767. doi:10.1214/aos/1056562461.

Nocedal J, Wright SJ (2006). *Numerical Optimization.* 2nd edition. Springer-Verlag.

Plummer M (2003). "**JAGS**: A Program for Analysis of Bayesian Graphical Models Using Gibbs Sampling." In K Hornik, F Leisch, A Zeileis (eds.), *Proceedings of the 3rd International Workshop on Distributed Statistical Computing (DSC 2003).* Vienna, Austria. ISSN 1609-395X. URL http://www.ci.tuwien.ac.at/Conferences/DSC-2003/.

Plummer M, Best N, Cowles K, Vines K (2006). "**code**: Convergence Diagnosis and Output Analysis for MCMC." *R News*, **6**(1), 7–11. URL http://CRAN.R-project.org/doc/Rnews/.

R Core Team (2016). *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria. URL https://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.

Stan Development Team (2016). *Stan Modeling Language User's Guide and Reference Manual, Version 2.14.0.* URL http://mc-stan.org/.

Van Heesch D (2016). "**Doxygen**: Generate Documentation from Source Code." URL http://www.stack.nl/~dimitri/doxygen/.

Van Rossum G, *et al.* (2016). *Python Programming Language.* URL http://www.python.org/.

A. Developer process

A.1. Version control and source repository

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

A.2. Continuous integration

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

A.3. Testing framework

Stan includes extensive unit tests for low-level C++ code. Unit tests are implemented using the open-source **googletest** framework (Google 2016). These unit tests evaluate every function for accuracy of values and derivatives, as well as error checking. 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 **make** file also runs tests of all of the built-in models, including almost all of the BUGS example models. Models are tested for both convergence and posterior mean estimation to within MCMC standard error.

A.4. Builds

The build process for Stan is highly automated through a cross-platform series of **make** files. The top-level **make** file 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 **make** file 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 **make** file also supplies targets to build all of the documentation C++ API documentation is generated using the **doxygen** package (Van Heesch 2016). The Stan manual (Stan Development Team 2016) is typeset using the LaTeX package (Mittelbach, Goossens, Braams, Carlisle, and Rowley 2004).

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

B. Library dependencies

Stan's programming language translates to C++ code that depends on the following three external libraries.

B.1. Boost

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

B.2. Eigen

Stan's handling of matrices and linear algebra is implemented through the **Eigen** C++ template library (Guennebaud, Jacob *et al.* 2010). Eigen uses template metaprogramming to achieve state-of-the-art performance for matrix and linear algebra operations with a great deal of flexibility with respect to input types. Unfortunately, many of the expression templates that Eigen uses for efficient static analysis and lazy evaluation are short-circuited because of Stan's need to have mixed type operations (i.e., multiplying a constant predictor matrix of double values by a parameter vector of algorithmic differentiation values). To make up for this in some important cases, Stan has provided compound functions such as the quadratic form, which allow speedups of both the matrix operations and their derivatives compared to a direct implementation using Stan's built-in operators.

B.3. Sundials

Stan uses the **SUNDIALS** suite of nonlinear differential equation solvers for solving stiff differential equations (Hindmarsh, Brown, Grant, Lee, Serban, Shumaker, and Woodward 2005). The subpackage **CVODES** implements the backward differentiation formula (BDF) algorithm with sensitivities (Cohen and Hindmarsh 1996).

Affiliation:

Bob Carpenter
Department of Statistics
Columbia University

1255 Amsterdam Avenue
New York, NY 10027, United States of America
E-mail: carp@alias-i.com
URL: http://mc-stan.org/

Journal of Statistical Software http://www.jstatsoft.org/
published by the Foundation for Open Access Statistics http://www.foastat.org/
January 2017, Volume 76, Issue 1 *Submitted:* 2013-07-30
doi:10.18637/jss.v076.i01 *Accepted:* 2015-10-26