Stan Modeling Language

User's Guide and Reference Manual

Stan Development Team

Stan Version 2.1.0

Friday 20th December, 2013

Stan Development Team. 2013. *Stan Modeling Language: User's Guide and Reference Manual.* Version 2.1.0

Copyright © 2011–2013, Stan Development Team.

Contents

Preface

Why Stan?

We[1] did not set out to build Stan as it currently exists. We set out to apply full Bayesian inference to the sort of multilevel generalized linear models discussed in Part II of (Gelman and Hill, 2007). These models 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.

The models we wanted to fit turned out to be a challenge for current general-purpose software to fit. A direct encoding in BUGS or JAGS can grind these tools to a halt. Matt Schofield found his multilevel time-series regression of climate on tree-ring measurements wasn't converging after hundreds of thousands of iterations.

Initially, Aleks Jakulin spent some time working on extending the Gibbs sampler in the Hierarchical Bayesian Compiler (Daumé, 2007), which as its name suggests, is compiled rather than interpreted. But even an efficient and scalable implementation does not solve the underlying problem that Gibbs sampling does not fare well with highly correlated posteriors. We finally realized we needed a better sampler, not a more efficient implementation.

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 tuning of the proposals.

The Path to Stan

We were at the same time starting to hear more and more about Hamiltonian Monte Carlo (HMC) and its ability to overcome some of the the problems inherent in Gibbs sampling. Matt Schofield managed to fit the tree-ring data using a hand-coded implementation of HMC, finding it converged in a few hundred iterations.

---

[1]In Fall 2010, the "we" consisted of Andrew Gelman and his crew of Ph.D. students (Wei Wang and Vince Dorie), postdocs (Ben Goodrich, Matt Hoffman and Michael Malecki), and research staff (Bob Carpenter and Daniel Lee). Previous postdocs whose work directly influenced Stan included Matt Schofield, Kenny Shirley, and Aleks Jakulin. Jiqiang Guo joined as a postdoc in Fall 2011. Marcus Brubaker, a computer science postdoc at Toyota Technical Institute at Chicago, joined the development team in early 2012. Michael Betancourt, a physics Ph.D. about to start a postdoc at University College London, joined the development team in late 2012 after months of providing useful feedback on geometry and debugging samplers at our meetings. Yuanjun Gao, a statistics graduate student at Columbia, and Peter Li, an undergraduate student at Columbia, joined the development team in the Fall semester of 2012. Allen Riddel joined the development team in Fall of 2013 and is currently maintaining PyStan.

HMC appeared promising but was also problematic in that the Hamiltonian dynamics simulation requires the gradient of the log posterior. Although it's possible to do this by hand, it is very tedious and error prone. That's when we discovered reverse-mode algorithmic differentiation, which lets you write down a templated C++ function for the log posterior and automatically compute a proper analytic gradient up to machine precision accuracy in only a few multiples of the cost to evaluate the log probability function itself. 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 in the COIN-OR toolkit. But neither package supported very many special functions (e.g., probability functions, log gamma, inverse logit) or linear algebra operations (e.g., Cholesky decomposition) and were not easily and modularly extensible.

So we built our own reverse-mode algorithmic differentiation package. But once we'd built our own reverse-mode algorithmic differentiation package, the problem was that we could not just plug in the probability functions from a package like Boost because they weren't templated on all the arguments. We only needed algorithmic differentiation variables for parameters, not data or transformed data, and promotion is very inefficient in both time and memory. So we wrote our own fully templated probability functions.

Next, we integrated the Eigen C++ package for matrix operations and linear algebra functions. Eigen makes extensive use of expression templates for lazy evaluation and the curiously recurring template pattern to implement concepts without virtual function calls. But 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`. This is a problem we have yet to optimize away as of Stan version 1.3, but we have plans to extend Eigen itself to support heterogeneous matrix operator types.

At this point (Spring 2011), we were happily fitting models coded directly in C++ on top of the pre-release versions of the Stan API. Seeing how well this all worked, we set our sights on the generality and ease of use of BUGS. So we designed a modeling language in which statisticians could write their models in familiar notation that could be transformed to efficient C++ code and then compiled into an efficient executable program.

The next problem we ran into as we started implementing richer models is variables with constrained support (e.g., simplexes and covariance matrices). Although it is possible to implement HMC with bouncing for simple boundary constraints (e.g., positive scale or precision parameters), it's not so easy with more complex multivariate constraints. To get around this problem, we introduced typed variables and automatically transformed them to unconstrained support with suitable adjustments to the log probability from the log absolute Jacobian determinant of the inverse trans-

forms.

Even with the prototype compiler generating models, we still faced a major hurdle to ease of use. HMC requires two tuning parameters (step size and number of steps) and is very sensitive to how they are set. The step size parameter could be tuned during warmup based on Metropolis rejection rates, but the number of steps was not so easy to tune while maintaining detailed balance in the sampler. This led to the development of the No-U-Turn sampler (NUTS) (Hoffman and Gelman, 2011, 2013), which takes an ever increasing number of steps until the direction of the simulation turns around, then uses slice sampling to select a point on the simulated trajectory.

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. The 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. But we thought the efficiency of compilation might compensate for the lack of ideal fit to the problem.

We realized we were doing redundant calculations, so we wrote a vectorized form of the normal distribution for multiple variates with the same mean and scale, which sped things up a bit. At the same time, we introduced some simple template metaprograms to remove the calculation of constant terms in the log probability. These both improved speed, but not enough. Finally, we figured out how to both vectorize and partially evaluate the gradients of the densities using a combination of expression templates and metaprogramming. At this point, we are within a factor of two or so of a hand-coded gradient function.

Later, when we were trying to fit a time-series model, we found that normalizing the data to unit sample mean and variance sped up the fits by an order of magnitude. Although HMC and NUTS are rotation invariant (explaining why they can sample effectively from multivariate densities with high correlations), they are not scale invariant. Gibbs sampling, on the other hand, is scale invariant, but not rotation invariant.

We were still using a unit mass matrix in the simulated Hamiltonian dynamics. The last tweak to Stan before version 1.0 was to estimate a diagonal mass matrix during warmup; this has since been upgraded to a full mass matrix in version 1.2. Both these extensions go a bit beyond the NUTS paper on *arXiv*. Using a mass matrix sped up the unscaled data models by an order of magnitude, though it breaks the nice theoretical property of rotation invariance. The full mass matrix estimation has rotational invariance as well, but scales less well because of the need to invert the mass matrix once and then do matrix multiplications every leapfrog step.

Stan 2

It's been over a year since the initial release of Stan, and we have been overjoyed by the quantity and quality of models people are building with Stan. We've also been a bit overwhelmed by the volume of traffic on our user's list and issue tracker.

We've been particularly happy about all the feedback we've gotten about installation issues as well as bugs in the code and documentation. We've been pleasantly surprised at the number of such requests which have come with solutions in the form of a GitHub pull request. That certainly makes our life easy.

As the code base grew and as we became more familiar with it, we came to realize that it required a major refactoring (see, for example, (Fowler et al., 1999) for a nice discussion of refactoring). So while the outside hasn't changed dramatically in Stan 2, the inside is almost totally different in terms of how the HMC samplers are organized, how the output is analyzed, how the mathematics library is organized, etc.

We've also improved our optimization algorithm (BFGS) and its parameterization. We've added more compile-time and run-time error checking for models. We've added many new functions, including new matrix functions and new distributions. We've added some new parameterizations and managed to vectorize all the univariate distributions. We've increased compatibility with a range of C++ compilers.

We've also tried to fill out the manual to clarify things like array and vector indexing, programming style, and the I/O and command-line formats. Most of these changes are direct results of user-reported confusions. So please let us know where we can be clearer or more fully explain something.

Finally, we've fixed all the bugs which we know about. It was keeping up with the latter that really set the development time back, including bugs that resulted in our having to add more error checking.

Stan's Future

We're not done. There's still an enormous amount of work to do to improve Stan. Our to-do list is in the form of a Wiki on GitHub:

> `https://github.com/stan-dev/stan/wiki/To-Do-List`

We are gradually weaning ourselves off of the to-do list in favor of the GitHub issue tracker (see the next section for a link).

Two major features are on the short-term horizon for us after Stan 2. The first is a differential equation solver, which will allow fitting parameters of ordinary differential equations as part of model building (PKBUGS supplies this functionality for BUGS and it has been rolled into OpenBUGS). The second big project is Riemannian manifold Hamiltonian Monte Carlo (RMHMC). Both of these projects require us to put

the finishing touches on higher-order automatic differentiation. We also have a number of smaller projects in the works, including more improvements to the modeling language itself, such as a way to define and reuse functions and general matrix and array index slicing.

You Can Help

Please let us know if you have comments about this manual or suggestions for Stan. We're especially interested in hearing about models you've fit or had problems fitting with Stan. The best way to communicate with the Stan team about user issues is through the following user's group.

    `http://groups.google.com/group/stan-users`

For reporting bugs or requesting features, Stan's issue tracker is at the following location.

    `https://github.com/stan-dev/stan/issues`

One of the main reasons Stan is freedom-respecting, open-source software[2] is that we love to collaborate. We're interested in hearing from you if you'd like to volunteer to get involved on the development side. We have all kinds of projects big and small that we haven't had time to code ourselves. For developer's issues, we have a separate group.

    `http://groups.google.com/group/stan-dev`

To contact the project developers off the mailing lists, send email to

    `stan@mc-stan.org`

The Stan Development Team
Friday 20th December, 2013

---

[2]See Appendix A for more information on Stan's licenses and the licenses of the software on which it depends.

Acknowledgements

Institutions

Grants

Individuals

Bug Reports

We're really thankful to everyone who's had the patience to try to get Stan working and reported bugs. All the gory details are available from Stan's issue tracker at the following URL.

https://github.com/stan-dev/stan/issues

Stanislaw Ulam, namesake of Stan and co-inventor of Monte Carlo methods (Metropolis and Ulam, 1949), shown here holding the Fermiac, Enrico Fermi's physical Monte Carlo simulator for neutron diffusion.

Image from (Giesler, 2000).

Part I

Introduction

1.   Overview

This document is both a user's guide and a reference manual for Stan's probabilistic modeling language. This introductory chapter provides a high-level overview of Stan. The next chapter provides a hands-on quick-start guide showing how Stan works in practice. Installation instructions are in Appendix B. The remaining parts of this document include a practically-oriented user's guide for programming models and a detailed reference manual for Stan's modeling language and associated programs and data formats.

1.1.   Stan Interfaces

There are three interfaces for Stan that are supported as part of the Stan project. Models and their use are the same across the three interfaces, and this manual is the modeling language manual for all three interfaces. All of the interfaces share initialization, sampling and tuning controls, and roughly share posterior analysis functionality.

CmdStan

CmdStan allows Stan to be run from the command line. In some sense, CmdStan is the reference implementation of Stan. This manual currently doubles as the CmdStan documentation. In the near term, the CmdStan documentation will be broken out of this manual and given its own manual.

RStan

RStan is the R interface to Stan. The installation and getting started guide for RStan can be found on GitHub at:

    https://github.com/stan-dev/rstan/wiki/
    RStan-Getting-Started

PyStan

PyStan is the Python interface to Stan. The installation and getting started guide for PyStan can be found on Read the Docs at:

    https://pystan.readthedocs.org/en/latest/getting_started.
    html

1.2. Stan Programs

A Stan program defines a statistical model through a conditional probability function $p(\theta|y;x)$, where θ is a sequence of modeled unknown values (e.g., model parameters, latent variables, missing data, future predictions), y is a sequence of modeled known values, and x is a sequence of unmodeled predictors and constants (e.g., sizes, hyperparameters).

Stan programs consist of variable type declarations and statements. Variable types include constrained and unconstrained integer, scalar, vector, and matrix types, as well as (multidimensional) arrays of other types. Variables are declared in blocks corresponding to the variable's use: data, transformed data, parameter, transformed parameter, or generated quantity. Unconstrained local variables may be declared within statement blocks.

Statements in Stan are interpreted imperatively, so their order matters. Atomic statements involve the assignment of a value to a variable. Sequences of statements (and optionally local variable declarations) may be organized into a block. Stan also provides bounded for-each loops of the sort used in R and BUGS.

The transformed data, transformed parameter, and generated quantities blocks contain statements defining the variables declared in their blocks. A special model block consists of statements defining the log probability for the model.

Within the model block, BUGS-style sampling notation may be used as shorthand for incrementing an underlying log probability variable, the value of which defines the log probability function. The log probability variable may also be accessed directly, allowing user-defined probability functions and Jacobians of transforms.

1.3. Compiling and Running Stan Programs

A Stan program is first compiled to a C++ program by the Stan compiler `stanc`, then the C++ program compiled to a self-contained platform-specific executable. Stan can generate executables for various flavors of Windows, Mac OS X, and Linux.[1] Running the Stan executable for a model first reads in and validates the known values y and x, then generates a sequence of (non-independent) identically distributed samples $\theta^{(1)}, \theta^{(2)}, \ldots$, each of which has the marginal distribution $p(\theta|y;x)$.

1.4. Sampling

For continuous parameters, Stan uses Hamiltonian Monte Carlo (HMC) sampling (Duane et al., 1987; Neal, 1994, 2011), a form of Markov chain Monte Carlo (MCMC) sam-

---

[1]A Stan program may also be compiled to a dynamically linkable object file for use in a higher-level scripting language such as R or Python.

pling (Metropolis et al., 1953). Stan 1.0 does not do discrete sampling.[2] Chapter 11 discusses how finite discrete parameters can be summed out of models.

HMC accelerates both convergence to the stationary distribution and subsequent parameter exploration by using the gradient of the log probability function. The unknown quantity vector θ is interpreted as the position of a fictional particle. Each iteration generates a random momentum and simulates the path of the particle with potential energy determined the (negative) log probability function. Hamilton's decomposition shows that the gradient of this potential determines change in momentum and the momentum determines the change in position. These continuous changes over time are approximated using the leapfrog algorithm, which breaks the time into discrete steps which are easily simulated. A Metropolis reject step is then applied to correct for any simulation error and ensure detailed balance of the resulting Markov chain transitions (Metropolis et al., 1953; Hastings, 1970).

Standard HMC involves three "tuning" parameters to which its behavior is quite sensitive. Stan's samplers allow these parameters to be set by hand or set automatically without user intervention.

The first two tuning parameters set the temporal step size of the discretization of the Hamiltonian and the total number of steps taken per iteration (with their product determining total simulation time). Stan can be configured with a user-specified step size or it can estimate an optimal step size during warmup using dual averaging (Nesterov, 2009; Hoffman and Gelman, 2011, 2013). In either case, additional randomization may be applied to draw the step size from an interval of possible step sizes (Neal, 2011).

Stan can be set to use a specified number of steps, or it can automatically adapt the number of steps during sampling using the No-U-Turn (NUTS) sampler (Hoffman and Gelman, 2011, 2013).

The third tuning parameter is a mass matrix for the fictional particle. Stan can be configured to estimate a diagonal mass matrix or a full mass matrix during warmup; Stan will support user-specified mass matrices in the future. Estimating a diagonal mass matrix normalizes the scale of each element θ_k of the unknown variable sequence θ, whereas estimating a full mass matrix accounts for both scaling and rotation,[3] but is more memory and computation intensive per leapfrog step due to the underlying matrix operations.

---

[2] Plans are in place to add full discrete sampling in Stan 2.0. An intermediate step will be to allow forward sampling of discrete variables in the generated quantities block for predictive modeling and model checking.

[3] These estimated mass matrices are global, meaning they are applied to every point in the parameter space being sampled. Riemann-manifold HMC generalizes this to allow the curvature implied by the mass matrix to vary by position.

Convergence Monitoring and Effective Sample Size

Samples in a Markov chain are only drawn with the marginal distribution $p(\theta|y;x)$ after the chain has converged to its equilibrium distribution. There are several methods to test whether an MCMC method has failed to converge; unfortunately, passing the tests does not guarantee convergence. The recommended method for Stan is to run multiple Markov chains each with different diffuse initial parameter values, discard the warmup/adaptation samples, then split the remainder of each chain in half and compute the potential scale reduction statistic, $\hat{R}$ (Gelman and Rubin, 1992).

When estimating a mean based on M independent samples, the estimation error is proportional to $1/\sqrt{M}$. If the samples are positively correlated, as they typically are when drawn using MCMC methods, the error is proportional to $1/\sqrt{\text{ESS}}$, where ESS is the effective sample size. Thus it is standard practice to also monitor (an estimate of) the effective sample size of parameters of interest in order to estimate the additional estimation error due to correlated samples.

Bayesian Inference and Monte Carlo Methods

Stan was developed to support full Bayesian inference. Bayesian inference is based in part on Bayes's rule,

$$p(\theta|y;x) \propto p(y|\theta;x)\,p(\theta;x),$$

which, in this unnormalized form, states that the posterior probability $p(\theta|y;x)$ of parameters θ given data y (and constants x) is proportional (for fixed y and x) to the product of the likelihood function $p(y|\theta;x)$ and prior $p(\theta;x)$.

For Stan, Bayesian modeling involves coding the posterior probability function up to a proportion, which Bayes's rule shows is equivalent to modeling the product of the likelihood function and prior up to a proportion.

Full Bayesian inference involves propagating the uncertainty in the value of parameters θ modeled by the posterior $p(\theta|y;x)$. This can be accomplished by basing inference on a sequence of samples from the posterior using plug-in estimates for quantities of interest such as posterior means, posterior intervals, predictions based on the posterior such as event outcomes or the values of as yet unobserved data.

1.5. Optimization

Stan also supports optimization-based inference for models. Given a posterior $p(\theta|y)$, Stan can find the posterior mode θ^*, which is defined by

$$\theta^* = \text{argmax}_\theta\, p(\theta|y).$$

Here the notation argmax$_u$ $f(v)$ is used to pick out the value of v at which $f(v)$ is maximized.

If the prior is uniform, the posterior mode corresponds to the maximum likelihood estimate (MLE) of the parameters. If the prior is not uniform, the posterior mode is sometimes called the maximum a posterior (MAP) estimate. If parameters (typically hierarchical) have been marginalized out, it's sometimes called a maximum marginal likelihood (MML) estimate.

Inference with Point Estimates

The estimate θ^* is a so-called "point estimate," meaning that it summarizes the posterior distribution by a single point, rather than with a distribution. Of course, a point estimate does not, in and of itself, take into account estimation variance. Posterior predictive inferences $p(\tilde{y}|y)$ can be made using the posterior mode given data y as $p(\tilde{y}|\theta^*)$, but they are not Bayesian inferences, even if the model involves a prior, because they do not take posterior uncertainty into account. If the posterior variance is low and the posterior mean is near the posterior mode, inference with point estimates can be very similar to full Bayesian inference.

"Empirical Bayes"

Fitting point estimates of priors and then using them for subsequent inference is sometimes called "empirical Bayes" (see, e.g., (Efron, 2012)).[4] Typically these optimizations will be done using maximum marignal likelihood rather than posterior modes of a full model. Sometimes Empirical Bayes point estimates will be obtained using moment matching (see, e.g., the rat-tumor example in Chapter 5 of (Gelman et al., 2003)).

Experimental Feature

Stan's optimizers have not been as well tested as its samplers, so they are still considered an "experimental" feature. We would love to hear back about successess or failures users have with optimization.

---

[4]The scare quotes on "empirical Bayes" are because the approach is no more empirical than full Bayes. Empirical Bayes approaches merely ignore some posterior uncertainty to make inference more efficient computationally.

2. Getting Started

This chapter is designed to help users get acquainted with the overall design of the Stan language and calling Stan from the command line. Later chapters are devoted to expanding on the material in this chapter with full reference documentation. The content is identical to that found on the getting-started with the command-line documentation on the Stan home page, http://mc-stan.org/.

2.1. For BUGS Users

Appendix C describes some similarities and important differences between Stan and BUGS (including WinBUGS, OpenBUGs, and JAGS).

2.2. Installation

For information about supported versions of Windows, Mac and Linux platforms along with step-by-step installation instructions, see Appendix B.

2.3. Building Stan

Building Stan itself works the same way across platforms. To build Stan, first open a command-line terminal application. Then change directories to the directory in which Stan is installed (i.e., the directory containing the file named makefile).

```
> cd <stan-home>
```

Then make the library with the following make command

```
> make bin/libstan.a
```

then make the model parser and code generator with the following call, adjusting the 2 in –j2 to the number of CPU cores available.

```
> make –j2 bin/stanc
```

Warning: The make program may take 10+ minutes and consume 2+ GB of memory to build stanc. Compiler warnings, such as uname: not found, may be safely ignored.

   Finally, make the Stan output summary program with the following make command.

```
> make bin/print
```

Building libstan.a, bin/stanc, and bin/print needs to be done only once.

2.4. Compiling and Executing a Model

The rest of this quick-start guide explains how to code and run a very simple Bayesian model.

A Simple Bernoulli Model

The following simple model is available in the source distribution located at `<stan-home>` as

    src/models/basic_estimators/bernoulli.stan

The file contains the following model.

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

The model assumes the binary observed data `y[1],...,y[N]` are i.i.d. with Bernoulli chance-of-success `theta`. The prior on `theta` is `beta(1,1)` (i.e., uniform).

Implicit Uniform Priors

If no prior is specified for a parameter, it is implicitly given a uniform prior on its support. For parameters such as `theta` in the example, which are constrained to fall between 0 and 1, this produces a proper uniform distribution on the support of `theta`. Because Beta(1, 1) is the uniform distribution, the following sampling statement can be eliminated from the model without changing the log probability calculation.

    theta ~ beta(1,1);

For parameters with unbounded support, the implicit uniform prior is improper. Stan allows improper priors to be specified in models, but posteriors must be proper in order for sampling to succeed.

Constraints on Parameters

The variable `theta` is defined with lower and upper bounds, which constrain its value. Parameters with constrained support should always specify appropriate constraints in the parameter declaration; if the constraints are absent, sampling will either slow down or stop altogether based on whether the initial values satisfy the constraints.

Vectorizing Sampling Statements

Iterations of the model will be faster if the loop over sampling statements is *vectorized* by replacing

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

with the equivalent vectorized form,

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

Performance gains from vectorization are not because loops are slow in Stan, but because calls to sampling statements are slow. Vectorization allows multiple calls to a sampling statement to be replaced with a single call that can share common calculations for the log probability function, its gradients, and error checking. For more tips on optimizing the performance of Stan models, see Chapter 19.

Data Set

A data set of N = 10 observations is available in the file

```
src/models/basic_estimators/bernoulli.data.R
```

The content of the file is as follows.

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

This defines the contents of two variables, N and y, using an R-like syntax (see Chapter 6 for more information).

Generating and Compiling the Model

A single call to `make` will generate the C++ code for a model with a name ending in `.stan` and compile it for execution. This call will also compile the library `libstan.a` and the parser/code generator `stanc` if they have not already been compiled.

First, change directories to `<stan-home>`, the directory where Stan was unpacked that contains the file named `makefile` and a subdirectory called `src/`.

```
> cd <stan-home>
```

Then issue the following command:

```
> make src/models/basic_estimators/bernoulli
```

The command for Windows is the same, including the forward slashes.

The `make` command may be applied to files in locations that are not subdirectories issued from another directory as follows. Just replace the relative path `src/models/...` with the actual path.

The C++ generated for the model and its compiled executable form will be placed in the same directory as the model.

Sampling from the Model

The model can be executed from the directory in which it resides.

```
> cd src/models/basic_estimators
```

To execute sampling of the model under Linux or Mac, use

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

The `./` prefix before the executable is only required under Linux and the Mac when executing a model from the directory in which it resides.

For the Windows DOS terminal, the `./` prefix is not needed, resulting in the following command.

```
> bernoulli sample data file=bernoulli.data.R
```

Whether the command is run in Windows, Linux, or on the Mac, the output is the same. First, the parameters are echoed to the standard output, which shows up on the terminal as follows.

```
 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)
...
```

The ellipses (...) indicate that the output continues (as described below).
    After the configuration has been displayed a short timing warning is given.

```
...
Gradient evaluation took 4e-06 seconds
1000 transitions using 10 leapfrog steps per transition would take 0.04 seconds
Adjust your expectations accordingly!
...
```

Next, the sampler counts up the iterations in place, reporting percentage completed, ending as follows.

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

Sampler Output

Each execution of the model results in the samples from a single Markov chain being written to a file in comma-separated value (CSV) format. The default name of the output file is output.csv.

The first part of the output file just repeats the parameters as comments (i.e., lines beginning with the pound sign (#)).

```
# 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 = 355899897
# output
#   file = output.csv (Default)
#   diagnostic_file =  (Default)
```

```
#    refresh = 100 (Default)
...
```

This is then followed by a header indicating the names of the values sampled.

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

The first column gives the log probability. The next columns, here columns two through five, provide sampler-dependent information. For basic Hamiltonian Monte Carlo (HMC) and its adaptive variant No-U-Turn sampler (NUTS), the sampler-depedent parameters are described in the following table.

Sampler	Parameter	Description
HMC	accept_stat__	Metropolis acceptance probability
HMC	stepsize__	Integrator step size
HMC	int_time__	Total integration time
NUTS	accept_stat__	Metropolis acceptance probability averaged over samples in the slice
NUTS	stepsize__	Integrator step size
NUTS	treedepth__	Tree depth
NUTS	n_divergent__	Number of divergent iterations

The rest of the columns in the header correspond to model parameters, here just theta in the sixth column. The parameter name header is output before warmup begins.

The result of any adaptation taking place during warmup is output next after the parameter names.

```
...
# Adaptation terminated
# Step size = 1.81311
# Diagonal elements of inverse mass matrix:
# 0.415719
...
```

The default sampler is NUTS with an adapted step size and a diagonal inverse mass matrix. For the running example, the step size is 1.81311, and the inverse mass contains the single entry 0.415719 corresponding to the parameter theta.

Samples from each iteration are printed out next, one per line in columns corresponding to the headers. [1]

---

[1]There are repeated entries due to the Metropolis accept step in the No-U-Turn sampling algorithm.

```
...
-6.95293,0.945991,1.09068,1,0,0.335074
-6.92373,0.938744,1.09068,0,0,0.181194
-6.83655,0.934833,1.09068,1,0,0.304882
...
-7.01732,1,1.09068,0,0,0.348244
-8.96652,0.48441,1.09068,0,0,0.549066
-7.22574,1,1.09068,0,0,0.383089
```

The output ends with timing details,

```
...
#  Elapsed Time: 0.006811 seconds (Warm-up)
#                0.011645 seconds (Sampling)
#                0.018456 seconds (Total)
```

Summarizing Sampler Output

The command-line program `bin/print` will display summary information about the run (for more information, see Chapter 5). To run `print` on the output file generated for `bernoulli` on Linux or Mac, use

> `<stan-home>/bin/print output.csv`

where `<stan-home>` is the path to where Stan was unpacked. For Windows use back-slashes for the executable,

> `<stan-home>\bin\print output.csv`

The output of the command will display information about the run followed by infor-mation for each parameter and generated quantity. For `bernoulli`, we ran 1 chain and saved 1000 iterations. The information is echoed to the standard output stream. For the running example, the path to `<stan-home>` can be specified from the direc-tory in which the Bernoulli model resides using `../` (with backslashes on Windows) as

> `../../../bin/print output.csv`

For Windows, reverse the slashes. The output is

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

Warmup took (0.0066) seconds, 0.0066 seconds total
Sampling took (0.011) seconds, 0.011 seconds total
```

	Mean	MCSE	StdDev	5%	50%	95%	N_Eff	N_Eff/s	R_hat
lp__	-7.3	3.5e-02	6.9e-01	-8.7e+00	-7.0	-6.7	390	34020	1.00
accept_stat__	0.64	1.2e-02	3.6e-01	5.1e-03	0.74	1.0	882	76898	1.00
stepsize__	1.8	7.8e-15	5.6e-15	1.8e+00	1.8	1.8	0.50	44	1.00
treedepth__	0.076	8.6e-03	2.7e-01	0.0e+00	0.00	1.0	942	82167	1.00
n_divergent__	0.00	0.0e+00	0.0e+00	0.0e+00	0.00	0.00	1000	90909	1.00
theta	0.25	4.2e-03	1.2e-01	9.0e-02	0.23	0.47	827	72146	1.00

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).

In addition to the general information about the runs, `print` displays summary statistics for each parameter and generated quantity.

In the `bernoulli` model, there is a single parameter, `theta`. The mean, standard error of the mean, standard deviation, the 5%, 50%, and 95% quantiles, number of effective samples (total and per second), and $\hat{R}$ value are displayed. These quantities and their uses are described in detail in Chapter 48.

The command `bin/print` can be called with more than one csv file by separating filenames with spaces. It will also take wildcards in specifying filenames. A typical usage of Stan from the command line would first create one or more Markov chains by calling the model executable, typically in parallel, writing the output CSV file for each into its own directory. Next, after all of the processes are finished, the results would be analyzed using `print` to assess convergence and inspect the means and quantiles of the fitted variables. Additionally, downstream inferences may be performed using the samples (e.g., to make decisions or predictions for unseen data).

Optimization

Stan can be used for finding posterior modes as well as sampling from the posterior. The model does not need to be recompiled in order to switch from optimization to sampling, and the data input format is the same. Although many command-line arguments may be provided to configure the optimizer, the following minimal command suffices, using defaults for everything but where to find the data file.

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

which prints out

```
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 = 2907588507
 output
   file = output.csv (Default)
   append_sample = 0 (Default)
   diagnostic_file =  (Default)
   append_diagnostic = 0 (Default)
   refresh = 100 (Default)

initial log joint probability = -10.9308
    Iter      log prob        ||dx||      ||grad||     alpha  # evals  Notes
       7      -5.00402   3.67055e-07   3.06339e-11        1       10
Optimization terminated normally:
  Convergence detected: change in objective function was below
  tolerance
```

The first part of the output reports on the configuration used, here indicating the default BFGS optimizer, with default initial stepsize and tolerances for monitoring convergence. The second part of the output indicates how well the algorithm fared, here converging and terminating normally. The numbers reported indicate that it took 7 iterations and 10 gradient evaluations, resulting in a final state state where the change in parameters was roughly 3.7e-7 and the length of the gradient roughly 3e-11. The `alpha` value is for step size used. This is, not surprisingly, far fewer iterations than required for sampling; even fewer iterations would be used with less stringent user-specified convergence tolerances.

Output from Optimization

The output from optimization is written into the file `output.csv` by default. The output follows the same pattern as the output for sampling, first dumping the entire set of parameters used.

```
# 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 = 2907588507
# output
#   file = output.csv (Default)
#   append_sample = 0 (Default)
#   diagnostic_file =  (Default)
#   append_diagnostic = 0 (Default)
#   refresh = 100 (Default)
lp__,theta
-5.00402,0.2000000000030634
```

Note that everything is a comment other than a line for the header, and a line for the values. Here, the header indicates the unnormalized log probability with lp__ and the model parameter theta. The maximum log probability is -5.0 and the posterior mode for theta is 0.20. The mode exactly matches what we would expect from the data. [2] Because the prior was uniform, the result 0.20 represents the maximum likelihood estimate (MLE) for the very simple Bernoulli model. Note that no uncertainty is reported.

Configuring Command-Line Options

The command-line options for running a model are detailed in Chapter 4. They can also be printed on the command line using Linux or Mac OS with

> *./bernoulli help-all*

and on Windows with

> *bernoulli help-all*

---

[2]The Jacobian adjustment included for the sampler's log probability function is not applied during optimization, because it can change the shape of the posterior and hence the solution.

It may help to glance at the command-line skeletons in Figure 4.4 through Figure 4.9 to get a handle on the options then read the detailed descriptions earlier in Chapter 4.

Testing Stan

To run the Stan unit tests of basic functionality, run the following commands from a shell (where <stan-home> is replaced top-level directory into which Stan was unpacked; it should contain a file named makefile).

```
> cd <stan-home>
> make -j4 O=0 test-unit
> make -j4 O=0 test-distributions
> make -j4 O=3 test-models
```

As before, -j4 indicates that four processes should be run in parallel; adjust the value 4 to correspond to the number of CPU cores available. Code optimization is specified by the letter 'O' followed by an equal sign followed by the digit '0' for no optimization and '3' for more optimization; optimization slows down compilation of the executable but reduces its execution time. Warnings can be safely ignored if the tests complete without a FAIL error.

Warning: The unit tests can take 30+ minutes and consume 3+ GB of memory with the default compiler, g++. The distribution test and model tests can take even longer. It is faster to run the Clang compiler (option CC=clang++), and to run in multiple processes in parallel (e.g., option -j4 for four threads).

Part II

Commands and Data Formats

3.   Compiling Stan Programs

Preparing a Stan program to be run involves two steps,

1. translating the Stan program to C++, and

2. compiling the resulting C++ to an executable.

This chapter discusses both steps, as well as their encapsulation into a single make target.

3.1.   Installing Stan

Before Stan can be run, it must be installed; see Appendix B for complete platform-specific installation details.

3.2.   Translating and Compiling through `make`

The simplest way to compile a Stan program is through the `make` build tool, which encapsulates the translation and compilation step into a single command. The commands making up the `make` target for compiling a model are described in the following sections, and the following chapter describes how to run a compiled model.

Translating and Compiling Test Models

There are a number of test models distributed with Stan which unpack into the path `src/models`. To build the simple example `src/models/basic_estimators/bernoulli.stan`, the following call to `make` suffices. First the directory is changed to Stan's home directory by replacing `<stan-home>` with the appropriate path.

```
> cd <stan-home>
```

The current directory should now contain the file named `makefile`, which is the default instructions used by `make`. From within the top-level Stan directory, the following call will build an executable form of the Bernoulli estimator.

```
> make src/models/basic_estimators/bernoulli
```

This will translate the model `bernoulli.stan` to a C++ file and compile that C++ file, putting the executable in `src/models/basic_distributions/bernoulli(.exe)`. Although the make command including arguments is itself portable, the target it

creates is different under Windows than in Unix-like platforms. Under Linux and the Mac, the executable will be called `bernoulli`, whereas under Windows it will be called `bernoulli.exe`.

Dependencies in `make`

A `make` target can depend on other `make` targets. When executing a `make` target, first all of the targets on which it depends are checked to see if they are up to date, and if they are not, they are rebuilt. This includes the top-level target itself. If the `make` target to build the Bernoulli estimator is invoked a second time, it will see that it is up to date, and not compile anything. But if one of the underlying files has changes since the last invocation `make`, such as the model specification file, it will be retranslated to C++ and recompiled to an executable.

There is a dependency included in the make target that will automatically build the `bin/stanc` compiler and the `bin/libstan.a` library whenever building a model.

Getting Help from the `makefile`

Stan's `makefile`, which contains the top-level instructions to `make`, provides extensive help in terms of targets and options. It is located at the top-level of the distribution, so first change directories to that location.

> `cd <stan-home>`

and then invoke make with the target `help`,

> `make help`

Options to `make`

Stan's `make` targets allow the user to change compilers, library versions for Eigen and Boost, as well as compilation options such as optimization.

These options should be placed right after the call to `make` itself. For instance, to specify the `clang++` compiler at optimization level 0, use

> `make CC=clang++ O=0 ...`

Compiler Option

The option `CC=g++` specifies the `g++` compiler and `CC=clang++` specifies the `clang++` compiler. Other compilers with other names may be specified the same way. A full path may be used, or just the name of the program if it can be found on the system execution path.

Optimization Option

The option O=0 (that's letter 'O', equal sign, digit '0'), specifies optimization level 0 (no optimization), whereas O=3 specifies optimization level 3 (effectively full optimization), with levels 1 and 2 in between.

With higher optimization levels, generated executable tends to be bigger (in terms of bytes in memory) and faster. For best results on computationally-intensive models, use optimization level 3 for the Stan library and for compiling models.

Library Options

Alternative versions of Eigen, Boost, and Google Test may be specified using the properties `EIGEN`, `BOOST`, and `GTEST`. Just set them equal to a path that resolves to an appropriate library. See the libraries distributed under `lib` to see which subdirectory of the library distribution should be specified in order for the include paths in the C++ code to resolve properly.

Additional `make` Targets

All of these targets are intended to be invoked from the top-level directory in which Stan was unpacked (i.e., the directory that contains the file named `makefile`).

Clean Targets

A very useful target is `clean-all`, invoked as

```
> make clean-all
```

This removes everything that's created automatically by `make`, including the `stanc` translator, the Stan libraries, and all the automatically generated documentation.

Make Target for `stanc`

To make the `stanc` compiler, use

```
> make bin/stanc
```

As with other executables, the executable `bin/stanc` will be created under Linux and Mac, whereas `bin/stanc.exe` will be created under Windows.

Make Target for Stan Library

To build the Stan library, use the following target,

```
> make bin/libstan.a
```

3.3. Translating Stan to C++ with `stanc`

Building the `stanc` Compiler and the Stan Library

Before the `stanc` compiler can be used, it must be built. Use the following command from the top-level distribution directory containing the file named `makefile`.

```
> make bin/stanc
```

This invocation produces the executable `bin/stanc` under Linux and Mac, and `bin/stanc.exe` under Windows. The invocation of `make`, including the forward slash, is the same on both platforms.

The default compiler option is `CC=g++` and the default optimization level is `O=3` (the letter 'O'); to see how to change these, see the previous section in this chapter on `make`.

The `stanc` Compiler

The `stanc` compiler converts Stan programs to C++ programs. The first stage of compilation involves parsing the text of the Stan program. If the parser is successful, the second stage of compilation generates C++ code. If the parser fails, it will provide a diagnostic error message indicating the location in the input where the failure occurred and reason for the failure.

The following example illustrates a fully qualified call to `stanc` to build the simple Bernoulli model; just replace `<stan-home>` with the top-level directory containing Stan (i.e., the directory containing the file named `makefile`).

For Linux and Mac:

```
> cd <stan-home>
> bin/stanc --name=bernoulli --o=bernoulli.cpp \
  src/models/basic_estimators/bernoulli.stan
```

The backslash (\) indicates a continuation of the same line.

For Windows:

```
> cd <stan-home>
> bin\stanc --name=bernoulli --o=bernoulli.cpp ^
    src/models/basic_estimators/bernoulli.stan
```

The caret (^) indicates continuation on Windows.

This call specifies the name of the model, here `bernoulli`. This will determine the name of the class implementing the model in the C++ code. Because this name is the name of a C++ class, it must start with an alphabetic character (a–z or A–Z) and

contain only alphanumeric characters (a-z, A-Z, and 0-9) and underscores (_) and should not conflict with any C++ reserved keyword.

The C++ code implementing the class is written to the file bernoulli.cpp in the current directory. The final argument, bernoulli.stan, is the file from which to read the Stan program.

Command-Line Options for stanc

The model translation program stanc is called as follows.

```
> stanc [options] model_file
```

The argument *model_file* is a path to a Stan model file ending in suffix .stan. The options are as follows.

--help
    Displays the manual page for stanc. If this option is selected, nothing else is done.

--version
    Prints the version of stanc. This is useful for bug reporting and asking for help on the mailing lists.

--name=*class_name*
    Specify the name of the class used for the implementation of the Stan model in the generated C++ code.
    Default: *class_name* = *model_file*_model

--o=*cpp_file_name*
    Specify the name of the file into which the generated C++ is written.
    Default: *cpp_file_name* = *class_name*.cpp

--no_main
    Include this flag to prevent the generation of a main function in the output.
    Default: generate a main function

3.4. Compiling C++ Programs

As shown in the previous section (Section 3.3), Stan converts a program in the Stan modeling language to a C++ program. This C++ program must then be compiled using a C++ compiler.

The C++ compilation step described in this chapter, the model translation step described in the last chapter, and the compilation of the dependent binaries bin/stanc and bin/libstan.a may be automated through make; see Section 3.2 for details.

Which Compiler?

Stan has been developed using two portable, open-source C++ compilers, `g++` and `clang++`, both of which run under and generate code for Windows, Macintosh, and Unix/Linux.[1]

The `clang++` compiler is almost twice as fast at low levels of optimization, but the machine code generated by `g++` at high optimization levels is faster.

What the Compiler Does

A C++ compiler like `g++` or `clang++` performs several lower-level operations in sequence,

1. parsing the input C++ source file(s),

2. generating (static or dynamically) relocatable object code, and

3. linking the relocatable object code into executable code.

These stages may be called separately, though the examples in this manual perform them in a single call. The compiler invokes the assembler to convert assembly language code to machine code, and the linker to resolve the location of references in the relocatable object files.

Compiler Optimization

Stan was written with an optimizing compiler in mind, which allows the code to be kept relatively clean and modular. As a result, Stan code runs as much as an order of magnitude or more faster with optimization turned on.

For development of C++ code for Stan, use optimization level 0; for sampling, use optimization level 3. These are controlled through Stan's makefile using `O=0` and directly through `clang++` or `g++` with `-O0`; in both cases, the first character is the letter 'O' and the second the digit '0'.

Building the Stan Library

Before compiling a Stan-generated C++ program, the Stan object library archive must be built using the makefile. This only needs to be done once and then the archive may be reused. The recommended build command for the Stan archive is as follows (replacing `<stan-home>` with the directory into which Stan was unpacked and which contains the file named `makefile`).

---

[1]As of the current version, Stan cannot be compiled using MSVC, the Windows-specific compiler from Microsoft. MSVC is able to compile the `stanc` compiler, but not the templates required for algorithmic differentiation and the Eigen matrix library.

```
> cd <stan-home>
> make CC=g++ O=3 bin/libstan.a
```

Please be patient and ignore the (unused function) warning messages. Compilation with high optimization on g++ takes time (as much as 10 minutes or more) and memory (as much as 3GB).

This example uses the g++ compiler for C++ (makefile option CC=g++). The clang++ compiler may be used by specifying CC=clang++.

This example uses compiler optimization level 3 (makefile option O=3). Turning the optimization level down to 0 allows the code to built in under a minute in less than 1GB of memory. This will slow down sampling as much as an order of magnitude or more, so it is not recommended for running models. It can be useful for working on Stan's C++ code.

Compiling a Stan Model

Suppose following the instructions in the last chapter (Section 3.3) that a Stan program has been converted to a C++ program that resides in the source file <stan-home>/my_model.cpp.

The following commands will produce an executable in the file my_model in the current working directory (<stan-home>).

```
> cd <stan-home>
> g++ -O3 -Lbin -Isrc -isystem lib/boost_1.54.0                    \
      -isystem lib/eigen_3.2.0 my_model.cpp -o my_model -lstan
```

The backslash (\) is used to indicate that the command is continued; it should be entered all one one line. The options used here are as follows.

-O3 sets optimization level 3,

-Lbin specifies that the archive is in the bin directory,

-Isrc specifies that the directory src should be searched for code (it contains the top-level Stan headers),

-isystem lib/boost_1.54.0 specifies the include directory for the Boost library,

-isystem lib/eigen_3.2.0 specifies the include directory for the Eigen library,

my_model.cpp specifies the name of the source file to compile, and

-o my_model is the name of the resulting executable produced by the command (suffixed by .exe in Windows).

> `-lstan` specifies the name of the archived library (not the name of the
> file in which it resides),

The library binary and source specifications are required, as is the name of the C++ file to compile. User-supplied directories may be included in header or archive form by specifying additional `-L`, `-l`, and `-I` options.

A lower optimization level may be specified. If there is no executable name specified using the `-o` option, then the model is written into a file named `a.out`.

Library Dependencies

Stan depends on two open-source libraries,

1. the Boost general purpose C++ libraries, and

2. the Eigen matrix and linear algebra C++ libraries.

These are both distributed along with Stan in the directory `<stan-home>/lib`.

The code for Stan itself is located in the directory `<stan-home>/src`. Because not all of Stan is included in the archive `bin/libstan.a`, the `src` directory must also be included for compilation.

4.   Running a Stan Program

Once a Stan program is compiled, it can be run in many different ways. It can be used to sample or optimize parameters, or to diagnose a model. Before diving into the detailed configurations, the first section provides some simple examples.

4.1.   Getting Started by Example

Once a Stan program defining a model has been converted to a C++ program for that model (see Section 3.3) and the resulting C++ program compiled to a platform-specific executable (see Section 3.4), the model is ready to be run.

All of the Stan functionality is highly configurable from the command line; the options are defined later in this chapter. Each command option also has defaults, which are used in this section.

Sampling

Suppose the executable is in file `my_model` and the data is in file `my_data`, both in the current working directory. To generate samples from a data set using the default settings, use one of the following, depending on platform.

Mac OS and Linux

```
> ./my_model sample data file=my_data
```

Windows

```
> my_model sample data file=my_data
```

On both platforms, this command reads the data from file `my_data`, runs warmup tuning for 1000 iterations (the values of which are discarded), and then runs the fully-adaptive NUTS sampler for 1000 iterations, writing the parameter (and other) values to the file `samples.csv` in the current working directory. When no random number seed is specified, a seed is generated from the system time.

Sampling in Parallel

The previous example executes one chain, which can be repeated to generate multiple chains. However, users may want to execute chains in parallel on a multicore machine.

Mac OS and Linux

To sample four chains using a Bash shell on Mac OS or Linux, execute

```
> for i in {1..4}                              \
  do                                           \
    ./my_model sample random seed=12345        \
       id=$i data file=my_data                 \
       output file=samples$i.csv refresh=0 & \
  done
```

The backslash (\) indicates that the big command continues on the next display line; the blank line at the end that returns control to the prompt. The ampersand (&) at the end of the nested command pushes each process into the background, so that the loop can continue without waiting for the current chain to finish. The id value makes sure that a non-overlapping set of random numbers are used for each chain. The refresh option being set to 0 turns off output; this is not necessary, but if output is used, it will be interleaved for all chains running concurrently. Also note that the output file is explicitly specified, with the variable $i being used to ensure the output file name for each chain is unique.

Windows

On Windows, the following is functionally equivalent to the Bash snippet above

```
> for /l %x in (1, 1, 4) do start /b model  sample   ^
    random seed=12345 id=%x data file=my_data         ^
    output file=samples%x.csv refresh=0
```

The caret (^) indicates a line continuation in DOS.

Combining Parallel Chains

Stan has commands to analyze the output of multiple chains, each stored in their own file; see Chapter 5. RStan also has commands to read in multiple CSV files produced by Stan's command-line sampler.

To compute posterior quantities, it is sometimes easier to have the chains merged into a single CSV file. If the grep and sed programs are installed, then the following will combine the four comma-separated values files into a single comma-separated values file. The command is the same on Windows, Mac OS and Linux.

```
> grep lp__ samples1.csv > combined.csv
> sed '/^[#l]/d' samples*.csv >> combined.csv
```

Scripting and Batching

The previous examples show how to sample in parallel from the command line. Operations like these can also be scripted, using shell scripts (`.sh`) on Mac OS and Linux and DOS batch (`.bat`) files on Windows. A sequence of several such commands can be executed from a single script file. Such scripts might contain `stanc` commands (see Section 3.3) and `bin/print` commands (see Chapter 5) can be executed from a single script file. At some point, it is worthwhile to move to something with stronger dependency control such as makefiles.

Optimization

Stan can find the posterior mode (assuming there is one). If the posterior is not convex, there is no guarantee Stan will be able to find the global mode as opposed to a local optimum of log probability.

For optimization, the mode is calculated without the Jacobian adjustment for constrained variables, which shifts the mode due to the change of variables. Thus modes correspond to modes of the model as written.

Windows

```
> my_model optimize data file=my_data
```

Mac OS and Linux

```
> ./my_model optimize data file=my_data
```

4.2. Diagnostics

Stan has a basic diagnostic feature that will calculate gradients of the initial state and compare them with those calculated with finite differences. If there are discrepancies, there is a problem with the model or initial states (or a bug in Stan). To run on the different platforms, use one of the following.

Windows

```
> my_model diagnose data file=my_data
```

Mac OS and Linux

```
> ./my_model diagnose data file=my_data
```

4.3. Command-Line Options

Stan executables are highly configurable, allowing the user to specify and customize not only the calculation method but also the data, output, initialization, and random number generation. The arguments are defined hierarchically so that, for example, optimization settings are not necessary when sampling.

The atomic elements of the hierarchy (i.e., those without corresponding values) are *categorical arguments* (sometimes called "flags") which define self-contained categories of arguments.

Stan's commands have more hierarchical structure than is typical of command line executables, which usually have at most two subgroups of commands. Arguments grouped within a category are not ordered with respect to each other. The only ordering is that the global options come before the method argument and subcommand-specific options after the method argument. For example, the following four commands all define the same configuration:[1]

```
> ./model sample output file=samples.csv             \
                    diagnostic_file=diagnostics.csv \
            random seed=1

> ./model sample output diagnostic_file=diagnostics.csv \
                    file=samples.csv                \
            random seed=1

> ./model sample random seed=1                       \
            output file=samples.csv                 \
                    diagnostic_file=diagnostics.csv

> ./model sample random seed=1                       \
            output diagnostic_file=diagnostics.csv \
                    file=samples.csv
```

The categorical arguments `output` and `random` can be in any order provided that the subarguments follow their respective parent, here `diagnostic_file` and `file` following `output` and `seed` coming after `random`. These four configurations exhaust all valid combinations.

Categorical arguments may appear is isolation, for example when introducing `sample` or `random`, or they may appear as the values for other arguments, such as

---

[1] The backslash (\) is used at the end of a line in a command to indicate that it continues on the next line. The indentation to indicate the structure of the command is for pedagogical purposes only; the same result would be obtained writing each command on one line with single spaces separating the elements.

`hmc` which not only introduces a category of HMC related arguments but also defines the value of the argument `algorithm`. A visual diagram of the available categorical arguments is shown in Figure 4.1, with the mutual exclusivity of these arguments as values shown in Figure 4.2. Specifying conflicting arguments causes the execution to immediately terminate.

Note that any valid argument configuration must either specify a method or a help request.

Method

All commands other than `help` must include at least one method, specified explicitly as `method=`*`method_name`* or implicitly with only *`method_name`*. Currently Stan supports the following methods:

Method	Description
sample	sample using MCMC
optimize	find posterior mode using optimization
diagnose	diagnose models

All remaining configurations are option, with default values provided for all arguments not explicitly specified.

Help

Informative output can be retrieved either globally, by requesting help at the top-level, or locally, by requesting help deeper into the hierarchy. Note that after any help has been displayed the execution immediately terminates, even if a method has been specified.

Top-Level Help

If `help` is specified as the only argument then a usage message is displayed. Similarly, specifying `help_all` by itself displays the entire argument hierarchy.

Context-Sensitive Help

Specifying `help` after any argument displays a description and valid options for that argument. For example,

```
./my_model sample help
```

```
id, data, init
```

```
random
       seed
```

```
output
       file, diagnostic_file, ...
```

```
method
```

```
  diagnose
        ...
```

```
  optimize
        ...
```

```
sample
       num_samples, num_warmup, save_warmup, thin
```

```
  adapt
        ...
```

```
  algorithm
```

```
    hmc
          ...
```

```
    rw_metropolis
            ...
```
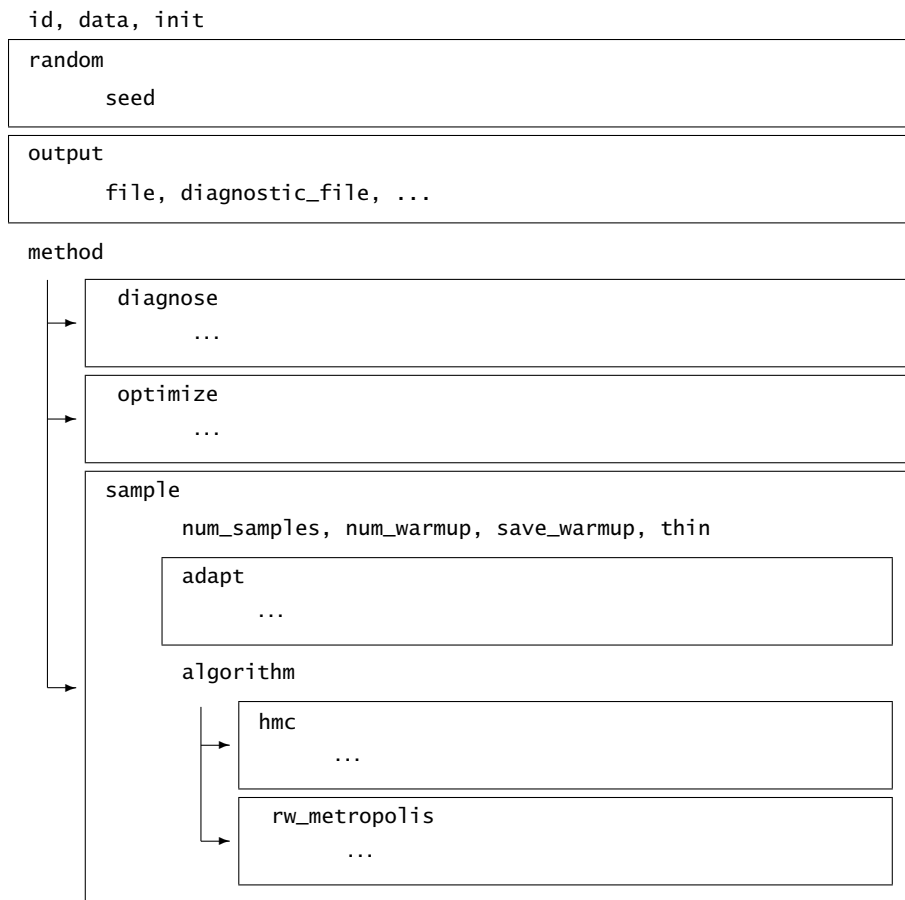
Figure 4.1: *In the hierarchical argument structure, certain arguments, such as* random *and* output, *introduce new categories of arguments. Categorical arguments may also appear as values of other arguments, such as* diagnose, optimize, *and* sample, *which define the mutually exclusive values for the argument* method.

```
id, data, init
```

```
random
        seed
```

```
output
        file, diagnostic_file, ...
```

```
method
```


Figure 4.2: *A valid argument configuration defines only one mutually exclusive argument.* *If conflicting arguments are specified, for example* method=optimize method=sample, *then* *execution immediately terminates with a warning message.*

provides the top-level options for the `sample` method.

Detailed information on the argument, and all arguments deriving from it, can accessed by specifying `help-all` instead,

```
./my_model sample help-all
```

4.4. Full Argument Hierarchy

Here we present the full argument hierarchy, along with relevant details. Some typical use-case examples are provided in the next section.

Typographical Conventions

The following typographical conventions are obeyed in the hierarchy.

- `arg=<value-type>`
  Arguments with values; displays the value type, legal values, and default value

- **`arg`**
  Isolated categorical arguments; displays all valid subarguments

- *`value`*
  Values; describes effect of selecting the value

- ***`avalue`***
  Categorical arguments that appear as values to other arguments; displays all valid subarguments

Top-Level Method Argument

Every command must have exactly one method specified as the very first argument. The value type of `list element` means that the valid values are enumerated as a list.

`method=<list element>`
  Analysis method (Note that `method=` is optional)
  Valid values: `sample`, `optimize`, `diagnose`
  (Defaults to `sample`)

Sampling-Specific Arguments

The following arguments are specific to sampling. The method argument `sample` (or `method=sample`) must come first in order to enable the subsequent arguments. The other arguments are optional and may appear in any order.

Figure 4.3: *Adaptation during warmup occurs in three stages: an initial fast adaptation interval (I), a series of expanding slow adaptation intervals (II), and a final fast adaptation interval (III). For HMC, both the fast and slow intervals are used for adapting the step size, while the slow intervals are used for learning the (co)variance necessitated by the metric. Iteration numbering starts at 1 on the left side of the figure and increases to the right.*

∟ **`sample`**
    Bayesian inference with Markov Chain Monte Carlo
    Valid subarguments: `num_samples`, `num_warmup`, `save_warmup`,
        `thin`, `adapt`, `algorithm`

∟ ∟ `num_samples=<int>`
    Number of sampling iterations
    Valid values: $0 \leq$ `num_samples`
    (Defaults to 1000)

∟ ∟ `num_warmup=<int>`
    Number of warmup iterations
    Valid values: $0 \leq$ `warmup`
    (Defaults to 1000)

∟ ∟ `save_warmup=<boolean>`
    Stream warmup samples to output?
    Valid values: `0, 1`
    (Defaults to 0)

∟ ∟ `thin=<int>`
    Period between saved samples
    Valid values: $0 <$ `thin`
    (Defaults to 1)

Sampling Adaptation-Specific Parameters

When adaptation is engaged the warmup period is split into three stages (Figure 4.3), with two *fast* intervals surrounding a series of growing *slow* intervals. Here fast and

slow refer to parameters that adapt using local and global information, respectively; the Hamiltonian Monte Carlo samplers, for example, define the step size as a fast parameter and the (co)variance as a slow parameter. The size of the the initial and final fast intervals and the initial size of the slow interval are all customizable, although user-specified values may be modified slightly in order to ensure alignment with the warmup period.

The motivation behind this partitioning of the warmup period is to allow for more robust adaptation. In the initial fast interval the chain is allowed to converge towards the typical set,[2] with only parameters that can learn from local information adapted. After this initial stage parameters that require global information, for example (co)variances, are estimated in a series of expanding, memoryless windows; often fast parameters will be adapted here as well. Lastly the fast parameters are allowed to adapt to the final update of the slow parameters.

Currently all Stan sampling algorithms utilize dual averaging to optimize the step size (this optimization during adaptation of the sampler should not be confused with running Stan's optimization method). This optimization procedure is extremely flexible and for completeness we have exposed each option, using the notation of (Hoffman and Gelman, 2011, 2013). In practice the efficacy of the optimization is sensitive to the value of these parameters, and we do not recommend changing the defaults without experience with the dual averaging algorithm. For more information, see the discussion of dual averaging in (Hoffman and Gelman, 2011, 2013).

Variances or covariances are estimated using Welford accumulators to avoid a loss of precision over many floating point operations.

The following subarguments are introduced by the categorical argument `adapt`. Each subargument must contiguously follow `adapt`, though they may appear in any order.

∟ ∟ **adapt**
   Warmup Adaptation
   Valid subarguments: `engaged`, `gamma`, `delta`, `kappa`, `t0`

∟ ∟ ∟ `engaged=<boolean>`
   Adaptation engaged?
   Valid values: `0`, `1`
   (Defaults to `1`)

∟ ∟ ∟ `gamma=<double>`
   Adaptation regularization scale

---

[2]The typical set is a concept borrowed from information theory and refers to the neighborhood (or neighborhoods in multimodal models) of significant posterior probability mass through which the Markov chain will travel in equilibrium.

Valid values: 0 < `gamma`
(Defaults to `0.05`)

ㄴ ㄴ ㄴ `delta=`*`<double>`*
Adaptation target acceptance statistic
Valid values: 0 < `delta` < 1
(Defaults to `0.8`)

ㄴ ㄴ ㄴ `kappa=`*`<double>`*
Adaptation relaxation exponent
Valid values: 0 < `kappa`
(Defaults to `0.75`)

ㄴ ㄴ ㄴ `t0=`*`<double>`*
Adaptation iteration offset
Valid values: 0 < `t0`
(Defaults to `10`)

ㄴ ㄴ ㄴ `init_buffer=`*`<unsigned int>`*
Width of initial fast adaptation interval
Valid values: All
(Defaults to `75`)

ㄴ ㄴ ㄴ `term_buffer=`*`<unsigned int>`*
Width of final fast adaptation interval
Valid values: All
(Defaults to `50`)

ㄴ ㄴ ㄴ `window=`*`<unsigned int>`*
Initial width of slow adaptation interval
Valid values: All
(Defaults to `25`)

By setting the acceptance statistic `delta` to a value closer to 1 (its value must be strictly less than 1 and its default value is 0.8), adaptation will be forced to use smaller step sizes. This can improve sampling efficiency (effective samples per iteration) at the cost of increased iteration times. Raising the value of `delta` will also allow some models that would otherwise get stuck overcome their blockages; see also the `stepsize_jitter` argument.

Sampling Algorithm- and Engine-Specific Arguments

The following batch of arguments are used to control the sampler used for sampling. The top-level specification is for engine, the only valid value of which is `hmc` (this will

change in the future as we add new samplers).

⌐ ⌐ `algorithm=<list element>`
  Sampling algorithm
  Valid values: `hmc`
  (Defaults to `hmc`)

Hamiltonian Monte Carlo is a very general approach to sampling that utilizes techniques of differential geometry and mathematical physics to generate efficient MCMC transitions. This generality manifests in a wealth of implementation choices.

⌐ ⌐ ⌐ ***hmc***
  Hamiltonian Monte Carlo
  Valid subarguments: `engine, metric, stepsize, stepsize_jitter`

All HMC implementations require at least two parameters: an integration step size and a total integration time. We refer to different specifications of the latter as *engines*.

In the `static_hmc` implementation the total integration time must be specified by the user, where as the `nuts` implementation uses the No-U-Turn Sampler to determine an optimal integration time dynamically.

⌐ ⌐ ⌐ ⌐ `engine=<list element>`
  Engine for Hamiltonian Monte Carlo
  Valid values: `static, nuts`
  (Defaults to `nuts`)

The following options are activated for static HMC.

⌐ ⌐ ⌐ ⌐ ⌐ ***static***
  Static integration time
  Valid subarguments: `int_time`

⌐ ⌐ ⌐ ⌐ ⌐ ⌐ `int_time=<double>`
  Total integration time for Hamiltonian evolution
  Valid values: $0 < $ `int_time`
  (Defaults to 2π)

These options are for NUTS, an adaptive version of HMC.

⌐ ⌐ ⌐ ⌐ ⌐ ***nuts***
  The No-U-Turn Sampler
  Valid subarguments: `max_depth`

Tree Depth

The `max_depth` argument for NUTS controls how much deeper than the initial node exploration is allowed to go. The tree depth actually used is calculated dynamically by NUTS each iteration, and reported along with the parameters in the output as `treedepth__`.

Each extension of the depth requires doubling the number of states and doubling the number of leapfrog steps. Each leapfrog step requires a gradient evaluation, and this is where the vast majority of time is spent evaluating using HMC. When NUTS has extended the tree `max_depth` layers, it stops automatically. If a model is hitting the maximum depth, it's a sign of extreme curvature and a strong warning sign there might be something wrong with the model. For most efficient sampling, the tree depth should not be constrained, otherwise HMC reverts to a random walk.

Stan currently counts depth starting from -1 rather than from the traditional 0, with a depth of -1 indicating no progress was made from the initial state due to slice rejection or constraint violation. [3] A depth of 0 means that one additional parameter state was considered.

See (Hoffman and Gelman, 2011, 2013) for more information on how NUTS works.

       `max_depth=<int>`
     Maximum tree depth
     Valid values: $0 <$ `max_depth`
     (Defaults to 10)

Euclidean Metric

All HMC implementations in Stan utilize quadratic kinetic energy functions which are specified up to the choice of a symmetric, positive-definite matrix known as a *mass matrix* or, more formally, a *metric* (Betancourt and Stein, 2011).

If the metric is constant then the resulting implementation is known as *Euclidean* HMC. Stan allows for three Euclidean HMC implementations: a unit metric, a diagonal metric, and a dense metric. These can be specified with the values `unit_e`, `diag_e`, and `dense_e`, respectively.

Future versions of Stan will also include dynamic metrics associated with *Riemannian* HMC (Girolami and Calderhead, 2011; Betancourt, 2012).

     `metric=<list element>`
     Geometry of base manifold
     Valid values: `unit_e, diag_e, dense_e`
     (Defaults to `diag_e`)

---

[3] This numbering will be corrected soon, at which point the interpretation of this argument will change by 1 and this section of the documentation will be updated.

˪ ˪ ˪ ˪ ˪ *unit_e*
    Euclidean manifold with unit metric

˪ ˪ ˪ ˪ ˪ *diag_e*
    Euclidean manifold with diag metric

˪ ˪ ˪ ˪ ˪ *dense_e*
    Euclidean manifold with dense metric

Step Size and Jitter

All implementations of HMC also use numerical integrators requiring a step size. We also allow that step size to be "jittered" randomly during sampling to avoid any poor interactions with a fixed step size and regions of high curvature. The maximum amount of jitter is 1, which will cause step sizes to be selected in the range of 0 to twice the adapted step size. Low step sizes can get HMC samplers unstuck that would otherwise get stuck with higher step sizes. The downside is that jittering below the adapted value will increase the number of leapfrog steps required and thus slow down iterations, whereas jittering above the adapted value can cause premature rejection due to simulation error in the Hamiltonian dynamics calculation. See (Neal, 2011) for further discussion of step-size jittering.

˪ ˪ ˪ ˪ stepsize=<*double*>
    Step size for discrete evolution
    Valid values: 0 < stepsize
    (Defaults to 1)

˪ ˪ ˪ ˪ stepsize_jitter=<*double*>
    Uniformly random jitter of the stepsize, in percent
    Valid values: 0 ≤ stepsize_jitter ≤ 1
    (Defaults to 0)

Optimization-Specific Commands

The following arguments are for the top-level method `optimize`. They allow control of the optimization algorithm, and some of its configuration. The other arguments may appear in any order.

˪ *optimize*
    Point estimation
    Valid subarguments: algorithm, iter, save_iterations

```
└ └ algorithm=<list element>
```
Optimization algorithm
Valid values: `nesterov`, `bfgs`, `newton`
(Defaults to `bfgs`)

The following options are for the BFGS optimizer. BFGS is the default optimizer and also much faster than the other optimizers.

Convergence monitoring in BFGS is controlled by a number of tolerance values, any one of which being satisified causes the algorithm to terminate with a solution.

· The log probability is considered to have converged if

$$\left| \log p(\theta^{(i)}|y) - \log p(\theta^{(i-1)}|y) \right| < \texttt{tol\_obj}.$$

· The parameters are considered to have converged if

$$||\theta^{(i)} - \theta^{(i-1)}|| < \texttt{tol\_param}.$$

· The gradient is considered to have converged to 0 if

$$||\nabla_\theta \log p(\theta^{(i)}|y)|| < \texttt{tol\_grad}.$$

Here, i is the current iteration, $\theta^{(i)}$ is the value of the parameters at iteration i, y is the data, $p(\theta^{(i)}|y)$ is the posterior probability of $\theta^{(i)}$ up to a proportion, ∇_θ is the gradient operator with respect to θ, $|u|$ is absolute value (L1 norm) of u, and $||u||$ is vector length (L2 norm) of u.

The other command-line argument for BFGS is `init_alpha`, which is first step size to try on the initial iteration. If the first iteration takes a long time (and requires a lot of function evaluations) set `init_alpha` to be the roughly equal to the alpha used in that first iteration. `init_alpha` has a tiny default value, which is reasonable for many problems but might be too large or too small depending on the objective function and initialization. Being too big or too small just means that the first iteration will take longer (i.e., require more gradient evaluations) before the line search finds a good step length. It's not a critical parameter, but for optimizing the same model multiple times (as you tweak things or with different data) being able to change it can save some real time.

```
└ └ └ bfgs
```
BFGS with linesearch
Valid subarguments: `stepsize`

└ └ └ └ `init_alpha=<double>`
    Line search step size for first iteration
    Valid values: $0 <$ `init_alpha`
    (Defaults to `0.001`)

└ └ └ └ `tol_obj=<double>`
    Convergence tolerance on changes in objective function value
    Valid values: $0 <$ `tol_obj`
    (Defaults to `1e-8`)

└ └ └ └ `tol_grad=<double>`
    Convergence tolerance on the norm of the gradient
    Valid values: $0 <$ `tol_grad`
    (Defaults to `1e-8`)

└ └ └ └ `tol_param=<double>`
    Convergence tolerance on changes in parameter value
    Valid values: $0 <$ `tol_param`
    (Defaults to `1e-8`)

The following options are for the Nesterov optimizer.

└ └ └ *nesterov*
    Nesterov's accelerated gradient method
    Valid subarguments: `stepsize`

└ └ └ └ `stepsize=<double>`
    Step size for discrete evolution
    Valid values: $0 <$ `stepsize`
    (Defaults to `1`)

The following argument is for Newton's optimization method; there are currently no configuration parameters for Newton's method, and it is not recommended because of the slow Hessian calcuation involving finite differences.

└ └ └ *newton*
    Newton's method

The remaining arguments apply to all optimizers.

└ └ `iter=<int>`
    Total number of iterations
    Valid values: $0 <$ `iter`
    (Defaults to `2000`)

└ └ `save_iterations=<boolean>`
> Stream optimization progress to output?
> Valid values: `0, 1`
> (Defaults to 0)

Diagnostic-Specific Arguments

The following arguments are specific to diagnostics. As of now, the only diagnostic is gradients of the log probability function.

└ *diagnose*
> Model diagnostics
> Valid subarguments: `test`

└ └ `test=<list element>`
> Diagnostic test
> Valid values: `gradient`
> (Defaults to `gradient`)

└ └ └ *gradient*
> Check model gradient against finite differences Valid subarguments: `epsilon`, `error`

└ └ └ └ `epsilon=<real>`
> Finite difference step size
> Valid values: 0 < `epsilon`
> (Defaults to `1e-6`)

└ └ └ └ `error=<real>`
> Error threshold
> Valid values: 0 < `error`
> (Defaults to `1e-6`)

General-Purpose Arguments

The following arguments may be used with any of the previous configurations. They may come either before or after the other subarguments of the top-level method.

Process Identifier Argument

`id=<int>`
> Unique process identifier
> Valid values: 0 < `id`
> (Defaults to 0)

Input Data Arguments

`data`
   Input data options
   Valid subarguments: `file`

↳ `file=<string>`
   Input data file
   Valid values: Path to existing file
   (Defaults to empty path)

Initialization Arguments

Initialization is only applied to parameters defined in the parameters block. Any initial values supplied for transformed parameters or generated quantities are ignored.

`init=<string>`
   Initialization method:
   - real number $x > 0$ initializes randomly bewteen [-x, x];
   - 0 initializes to 0;
   - non-number interpreted as a data file
   Valid values: All
   (Defaults to 2)

Random Number Generator Arguments

`random`
   Random number configuration
   Valid subarguments: `seed`

↳ `seed=<unsigned int>`
   Random number generator seed
   Valid values:
   - `seed` ≥ 0 generates seed;
   - `seed` < 0 uses seed generated from time
   (Defaults to –1)

Output Arguments

`output`
   File output options
   Valid subarguments: `file, diagnostic_file,`
      `, refresh`

└ file=*<string>*
    Output file
    Valid values: Valid path
    (Defaults to `output.csv`)

└ diagnostic_file=*<string>*
    Auxiliary output file for diagnostic information
    Valid values: Valid path
    (Defaults to empty path)

└ refresh=*<int>*
    Number of interations between screen updates
    Valid values: 0 < `refresh`
    (Defaults to 100)

4.5.   Command-Line Option Examples

The hierarchical structure of the command-line options can be intimidating, and here we provide an example workflow to help ease the introduction to new users, especially those used to Stan 1.3 or earlier releases. The examples in this section are for Mac OS and Linux; on Windows, just remove the `./` before the executable and change the line-continuation character from Unix's \ to DOS's ^. As in previous sections, the indentation on continued lines is for pedagogical purposes only and does not convey any content to the executable.

Let's say that we've just built our model, `model`, and are ready to run. We begin by specifying data and init files,

```
> ./model data file=model.data.R init=model.init.R
```

but our model doesn't run. Instead, the above command prints

```
A method must be specified!
Failed to parse arguments, terminating Stan
```

The problem is that we forgot to specify a method.

All Stan arguments have default values, except for the method. This is the only argument that must be specified by the user and a model will not run without it (not to say that the model will run without error, for example a model that requires data will eventually fail unless an input file is specified with `file` under `data`). Assuming that we want to draw MCMC samples from our model, we can either specify a method implicitly,

```
> ./model sample data file=model.data.R init=model.init.R
```

or explicitly,

```
> ./model method=sample data file=model.data.R  \
         init=model.init.R
```

In either case our model now executes without any problem.

Now let's say that we want to customize our execution. In particular we want to set the seed for the random number generator, but we forgot the specific argument syntax. Information for each argument can displayed by calling `help`,

```
> ./model random help
```

which returns

```
random
  Random number configuration
  Valid subarguments: seed
...
```

before printing usage information. For information on the seed argument we just call help one level deeper,

```
> ./model random seed help
```

which returns

```
seed=<unsigned int>
  Random number generator seed
  Valid values: seed > 0, if negative seed is generated from time
  Defaults to -1
  ...
```

Fully informed, we can now run with a given seed,

```
> ./model method=sample data fle=model.data.R  \
         init=model.init.R                   \
         random seed=5
```

The arguments `method`, `data`, `init`, and `random` are all top-level arguments. To really see the power of a hierarchical argument structure let's try to drill down and specify the metric we use for HMC: instead of the default diagonal Euclidean metric, we want to use a dense Euclidean metric. Attempting to specify the metric we try

```
> ./model method=sample data file=model.data.R  \
          init=model.init.R                      \
          random seed=5                          \
          metric=unit
```

only to have the execution fail with the message

```
metric=unit_e is either mistyped or misplaced.
Perhaps you meant one of the following valid configurations?
  method=sample algorithm=hmc metric=<list_element>
Failed to parse arguments, terminating Stan
```

The argument `metric` does exist, but not at the top-level. In order to specify it we have to drill down into sample by first specifying the sampling algorithm, as noted in the suggestion,

```
> ./model method=sample algorithm=hmc metric=unit \
          data file=model.data.R                   \
          init=model.init.R                         \
          random seed=5
```

Unfortunately we still messed up,

```
unit is not a valid value for "metric"
  Valid values: unit_e, diag_e, dense_e
Failed to parse arguments, terminating Stan
```

Tweaking the metric name we make one last attempt,

```
> ./model method=sample algorithm=hmc metric=unit_e \
          data file=model.data.R                     \
          init=model.init.R                           \
          random seed=5
```

which successfully runs.

Finally, let's consider the circumstance where our model runs fine but the NUTS iterations keep saturating the default tree depth limit of 10. We need to change the limit, but how do we specify NUTS let alone the maximum tree depth? To see how let's take advantage of the `help-all` option which prints all arguments that derive from the given argument. We know that NUTS is somehow related to sampling, so we try

```
> ./model method=sample help-all
```

which returns the verbose output,

```
sample
  Bayesian inference with Markov Chain Monte Carlo
  Valid subarguments: num_samples, num_warmup,
                      save_warmup, thin, adapt, algorithm

  num_samples=<int>
    Number of sampling iterations
    Valid values: 0 <= num_samples
    Defaults to 1000

  num_warmup=<int>
    Number of warmup iterations
    Valid values: 0 <= warmup
    Defaults to 1000

  save_warmup=<boolean>
    Stream warmup samples to output?
    Valid values: [0, 1]
    Defaults to 0

  thin=<int>
    Period between saved samples
    Valid values: 0 < thin
    Defaults to 1

  adapt
    Warmup Adaptation
    Valid subarguments: engaged, gamma, delta, kappa, t0

    engaged=<boolean>
      Adaptation engaged?
      Valid values: [0, 1]
      Defaults to 1

    gamma=<double>
      Adaptation regularization scale
      Valid values: 0 < gamma
      Defaults to 0.05

    delta=<double>
      Adaptation target acceptance statistic
      Valid values: 0 < delta < 1
      Defaults to 0.65
```

```
  kappa=<double>
    Adaptation relaxation exponent
    Valid values: 0 < kappa
    Defaults to 0.75

  t0=<double>
    Adaptation iteration offset
    Valid values: 0 < t0
    Defaults to 10

algorithm=<list element>
  Sampling algorithm
  Valid values: hmc
  Defaults to hmc

  hmc
    Hamiltonian Monte Carlo
    Valid subarguments: engine, metric, stepsize,
                        stepsize_jitter

    engine=<list element>
      Engine for Hamiltonian Monte Carlo
      Valid values: static, nuts
      Defaults to nuts

      static
        Static integration time
        Valid subarguments: int_time

        int_time=<double>
          Total integration time for Hamiltonian evolution
          Valid values: 0 < int_time
          Defaults to 2 * pi

      nuts
        The No-U-Turn Sampler
        Valid subarguments: max_depth

        max_depth=<int>
          Maximum tree depth
          Valid values: 0 < max_depth
          Defaults to 10
```

```
        metric=<list element>
          Geometry of base manifold
          Valid values: unit_e, diag_e, dense_e
          Defaults to diag_e

          unit_e
            Euclidean manifold with unit metric

          diag_e
            Euclidean manifold with diag metric

          dense_e
            Euclidean manifold with dense metric

        stepsize=<double>
          Step size for discrete evolution
          Valid values: 0 < stepsize
          Defaults to 1

        stepsize_jitter=<double>
          Uniformly random jitter of the stepsize, in percent
          Valid values: 0 <= stepsize_jitter <= 1
          Defaults to 0
    ...
```

Following the hierarchy, the maximum tree depth derives from nuts, which itself is a value for the argument engine which derives from hmc. Adding this to our previous call we attempt

```
> ./model method=sample                      \
          algorithm=hmc                      \
              metric=unit_e                  \
              engine=nuts max_depth=-15      \
       data file=model.data.R                \
       init=model.init.R                     \
       random seed=5                         \
```

which yields

```
-1 is not a valid value for "max_depth"
  Valid values: 0 < max_depth
Failed to parse arguments, terminating Stan
```

Where did that negative sign come from? Clumsy fingers are nothing to be embarrassed about, especially with such complex argument configurations. Removing the guilty character, we try

```
> ./model method=sample                    \
            algorithm=hmc                   \
                metric=unit_e               \
                engine=nuts max_depth=15    \
        data file=model.data.R              \
        init=model.init.R                   \
        random seed=5
```

which finally runs without issue.

4.6. Command Templates

This section provides templates for all of the arguments deriving from each of the possible methods: sample, optimize, and diagnose. Arguments in square brackets are optional, those not in square brackets are required for the template.

Sampling Templates

The No-U-Turn sampler (NUTS) is the default (and recommended) sampler for Stan. The full set of configuration options is in Figure 4.4.

A standard Hamiltonian Monte Carlo (HMC) sampler with user-specified integration time may also be used. Its set of configuration options are shown in Figure 4.5.

Both NUTS and HMC may be configured with either a unit, diagonal or dense Euclidean metric, with a diagonal metric the default.[4] A unit metric provides no parameter-by-parameter scaling, a diagonal metric scales each parameter independently, and a dense metric also rotates the parameters so that correlated parameters may move together. Although dense metrics offer the hope of superior simulation performance, they require more computation per iteration. Specifically for m samples of a model with n parameters, the dense metric requires $\mathcal{O}(n^3 \log(m) + n^2 m)$ operations, whereas diagonal metrics require only $\mathcal{O}(n m)$. Furthermore, dense metrics are difficult to estimate, given the $\mathcal{O}(n^2)$ components with complex interdependence.

---

[4]In Euclidean HMC, a diagonal metric emulates different step sizes for each parameter. Explicitly varying step sizes were used in Stan 1.3 and before; Neal (2011) discusses the equivalence.

```
> ./my_model sample                                      \
            algorithm=hmc                                \
                engine=nuts                              \
                    [max_depth=<int>]                    \
                [metric={unit_e,diag_e,dense_e}]         \
                [stepsize=<double>]                      \
                [stepsize_jitter=<double>]               \
            [num_samples=<int>]                          \
            [num_warmup=<int>]                           \
            [save_warmup=<boolean>]                      \
            [thin=<int>]                                 \
            [adapt                                       \
                [engaged=<boolean>]                      \
                [gamma=<double>]                         \
                [delta=<double>]                         \
                [kappa=<double>]                         \
                [t0=<double>] ]                          \
        [data file=<string>]                             \
        [init=<string>]                                  \
        [random seed=<int>]                              \
        [output                                          \
            [file=<string>]                              \
            [diagnostic_file=<string>]                   \
            [refresh=<int>] ]
```

Figure 4.4: *Command skeleton for invoking the no-U-turn sampler (NUTS). This is the same skeleton as that for basic HMC in Figure 4.5. Elements in braces are optional. All arguments and their default values are described in detail in Section 4.4.*

```
> ./my_model sample                                    \
         algorithm=hmc                                 \
             engine=static                             \
               [int_time=<double>]                     \
             [metric={unit_e,diag_e,dense_e}]          \
             [stepsize=<double>]                       \
             [stepsize_jitter=<double>]                \
         [num_samples=<int>]                           \
         [num_warmup=<int>]                            \
         [save_warmup=<boolean>]                       \
         [thin=<int>]                                  \
         [adapt                                        \
             [engaged=<boolean>]                       \
             [gamma=<double>]                          \
             [delta=<double>]                          \
             [kappa=<double>]                          \
             [t0=<double>] ]                           \
     [data file=<string>]                              \
     [init=<string>]                                   \
     [random seed=<int>]                               \
     [output                                           \
         [file=<string>]                               \
         [diagnostic_file=<string>]                    \
         [refresh=<int>] ]
```

Figure 4.5: *Command skeleton for invoking the basic Hamiltonian Monte Carlo sampler (HMC). This is the same as the NUTS command skeleton shown in Figure 4.4 other than for the engine. Elements in braces are optional. All arguments and their default values are described in detail in Section 4.4.*

```
> ./my_model optimize                           \
            algorithm=bfgs                       \
                [init_alpha=<double>]            \
                [tol_obj=<double>]               \
                [tol_grad=<double>]              \
                [tol_param=<double>]             \
            [iter=<int>]                         \
            [save_iterations=<boolean>]          \
        [data file=<string>]                     \
        [init=<string>]                          \
        [random seed=<int>]                      \
        [output                                  \
            [file=<string>]                      \
            [diagnostic_file=<string>]           \
            [refresh=<int>] ]
```

Figure 4.6: *Command skeleton for invoking the BFGS optimizer. All arguments and their default values are described in detail in Section 4.4.*

Optimization Templates

Stan supports several optimizers. These share many of their configuration options with the samplers. The default optimizer is the Broyden-Fletcher-Goldfarb-Shanno (BFGS) method (see (Nocedal and Wright, 2006) for more information on BFGS). The command skeleton for BFGS is in Figure 4.6.

Stan also supports Nesterov's dual-averaging method (Nesterov, 2009) for optimization. The BFGS method is the default approach because it is more efficient. The command skeleton for dual averaging is shown in Figure 4.7; it is identical to that for BFGS other than the algorithm name. Stan also supports Newton's method; see (Nocedal and Wright, 2006) for more information. This method is the least efficient of the three, but has the advantage of setting its own step size. Other than not having a stepsize argument, the skeleton for Newton's method shown in Figure 4.8 is identical to that for BFGS and Nesterov's dual averaging.

Diagnostic Command Skeleton

Stan reports on gradients for the model at a specified or randomly generated initial value. The command-skeleton in this case is very simple, and shown in Figure 4.9.

```
> ./my_model optimize                           \
            algorithm=nesterov                  \
              [stepsize=<double>]               \
            [iter=<int>]                         \
            [save_iterations=<boolean>]          \
        [data file=<string>]                    \
        [init=<string>]                         \
        [random seed=<int>]                     \
        [output                                 \
            [file=<string>]                     \
            [diagnostic_file=<string>]          \
            [refresh=<int>] ]
```

Figure 4.7: *Command skeleton for invoking the Nesterov dual-averaging ptimizer. All arguments and their default values are described in detail in Section 4.4.*

```
> ./my_model optimize                           \
            algorithm=newton                    \
            [iter=<int>]                         \
            [save_iterations=<boolean>]          \
        [data file=<string>]                    \
        [init=<string>]                         \
        [random seed=<int>]                     \
        [output                                 \
            [file=<string>]                     \
            [diagnostic_file=<string>]          \
            [refresh=<int>] ]
```

Figure 4.8: *Command skeleton for invoking the Newton optimizer. All arguments and their default values are described in detail in Section 4.4.*

```
> ./my_model diagnose                   \
            [test=gradient]             \
                [epsilon=<real>]        \
                [error=<real>]          \
        [data file=<string>]            \
        [init=<string>]                 \
        [random seed=<int>]             \
```

Figure 4.9: *Command skeleton for invoking model diagnostics. All arguments and their default values are described in detail in Section 4.4.*
```

# 5.  Print Command for Output Analysis

Stan is distributed with a print command that is able to read in the output of one or more Markov chains and summarize the posterior fits. This operation mimics the `print(fit)` command in RStan, which itself was modeled on the print functions from R2WinBUGS and R2jags.

## 5.1.  Building the Print Command

Stan's `print` command is built along with `stanc` into the `bin` directory. It can be compiled directly using the makefile as follows from the home directory into which Stan was unpacked (here written as `<stan-home>`).

```
> cd <stan-home>
> make bin/print
```

All the usual compiler options from Stan's makefile apply, such as `O=N` to set optimization level to N, and `CC=clang++` to set the compilation to use clang.

5.2. Running the Print Command

The print command is executed on one or more samples.csv files. These files may be provided as command-line arguments separated by spaces. That means that wildcards may be used, as they will be replaced by space-separated file names by the operating system's command-line interpreter.

Suppose there are three samples files in a directory generated by fitting a negative binomial model to a small data set.

```
> ls samples*.csv

samples1.csv          samples2.csv          samples3.csv

> bin/print samples*.csv
```

The result of `bin/print` is displayed in Figure 5.1.[1] The posterior is skewed to the high side, resulting in posterior means ($\alpha = 17$ and $\beta = 10$) that are a long way away from the posterior medians ($\alpha = 9.5$ and $\beta = 6.2$); the posterior median is the value listed under 50%, which is the 50th percentile of the posterior values.

For Windows, the forward slash in paths need to be converted to backslashes.

---

[1]RStan's and PyStan's output analysis print may be different than that in the command-line version of Stan.

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

Warmup took (0.054) seconds, 0.054 seconds total
Sampling took (0.059) seconds, 0.059 seconds total

                Mean     MCSE   StdDev     5%    50%    95%  N_Eff  N_Eff/s  R_hat
lp__             -14  6.2e-02  1.0e+00    -16    -14    -13    283     4773   1.00
accept_stat__   0.88  5.6e-03  1.8e-01   0.51   0.95    1.0   1000    16881   1.00
stepsize__      0.30  1.3e-15  8.9e-16   0.30   0.30   0.30   0.50      8.5   1.00
treedepth__      1.4  2.6e-02  8.0e-01   0.00    1.0    2.0    946    15978   1.00
n_divergent__    1.4  0.0e+00  0.0e+00   0.00    0.0    0.0   1000    16949   1.00
alpha             17  1.8e+00  2.5e+01    1.9    9.5     50    181     3054   1.00
beta              10  1.1e+00  1.4e+01    1.2    6.2     31    181     3057   1.00

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 5.1: *Example output from* bin/print. *The model parameters are* alpha *and* beta. *The values for each quantity are the posterior means, standard deviations, and quantiles, along with Monte-Carlo standard error, effective sample size estimates (per second), and convergence diagnostic statistic. These values are all estimated from samples. In addition to the parameters, the value* lp__ *is the total log probability computed by the model (up to an additive constant). The quantity* accept_stat__ *is the NUTS acceptance statistic used by NUTS for slice and Metropolis rejection,* stepsize__ *the step size used by NUTS in its Hamiltonian simulation, and* treedepth__ *is the depth of tree used by NUTS, which is the log (base 2) of the number of leapfrog steps taken during the Hamiltonian simulation.* n_divergent__ *gives the number of leapfrog iterations with diverging error; because NUTS terminates at the first divergent iteration this should always be either 0 or 1.*

5.3.   Command-line Options

In addition to the filenames, `print` includes three flags to customize the output.

`help`
> Prints usage information
> No help output by default

`sig_figs=<int>`
> Sets the number of significant figures displayed in the output
> Valid values: 0 <sig_figs
> (default = 2)

`autocorr=<int>`
> Calculates and then displays the autocorrelation of the specified chain
> Valid values: Any integer matching a chain index
> (No autocorrelation output by default)

6.   Dump Data Format

For representing structured data in files, Stan uses the dump format introduced in S and used in R and JAGS (and in BUGS, but with a different ordering). A dump file is structured as a sequence of variable definitions. Each variable is defined in terms of its dimensionality and its values. There are three kinds of variable declarations, one for scalars, one for sequences, and one for general arrays.

6.1.   Creating Dump Files

Dump files can be created from R using RStan. The function is `stan_rdump` in package `rstan`.

   Using R's native `dump()` function can produce dump files which Stan cannot read in. The underlying cause is that R gets creative in the format it uses for output, only being constrained to something that can be executed in R. So it will write the array containing the values 1, 2, 3, 4 as `1:4` rather than as `c(1,2,3,4)`.

6.2.   Scalar Variables

A simple scalar value can be thought of as having an empty list of dimensions. Its declaration in the dump format follows the S assignment syntax. For example, the following would constitute a valid dump file defining a single scalar variable `y` with value 17.2.

```
y <-
17.2
```

A scalar value is just a zero-dimensional array value.

6.3.   Sequence Variables

One-dimensional arrays may be specified directly using the S sequence notation. The following example defines an integer-value and a real-valued sequence.

```
n <- c(1,2,3)
y <- c(2.0,3.0,9.7)
```

Arrays are provided without a declaration of dimensionality because the reader just counts the number of entries to determine the size of the array.

Sequence variables may alternatively be represented with R's colon-based notation. For instance, the first example above could equivalently be written as

```
n <- 1:3
```

The sequence denoted by `1:3` is of length 3, running from 1 to 3 inclusive. The colon notation allows sequences going from high to low, as in the first of the following examples, which is equivalent to the second.

```
n <- 2:-2
n <- c(2,1,0,-1,-2)
```

6.4. Array Variables

For more than one dimension, the dump format uses a dimensionality specification. For example,

```
y <- structure(c(1,2,3,4,5,6), .Dim = c(2,3))
```

This defines a 2×3 array. Data is stored in column-major order, meaning the values for y will be as follows.

```
y[1,1] = 1     y[1,2] = 3     y[1,3] = 5
y[2,1] = 2     y[2,2] = 4     y[2,3] = 6
```

The `structure` keyword just wraps a sequence of values and a dimensionality declaration, which is itself just a sequence of non-negative integer values. The product of the dimensions must equal the length of the array.

If the values happen to form a contiguous sequence of integers, they may be written with colon notation. Thus the example above is equivalent to the following.

```
y <- structure(1:6, .Dim = c(2,3))
```

The same applies to the specification of dimensions, though it is perhaps less likely to be used. In the above example, c(2,3) could be written as `2:3`.

Arrays of more than two dimensions are written in a last-index major form. For example,

```
z <- structure(1:24, .Dim = c(2,3,4))
```

produces a three-dimensional `int` (assignable to `real`) array z with values

```
z[1,1,1] =  1   z[1,2,1] =  3   z[1,3,1] =  5
z[2,1,1] =  2   z[2,2,1] =  4   z[2,3,1] =  6

z[1,1,2] =  7   z[1,2,2] =  9   z[1,3,2] = 11
z[2,1,2] =  8   z[2,2,2] = 10   z[2,3,2] = 12

z[1,1,3] = 13   z[1,2,3] = 15   z[1,3,3] = 17
z[2,1,3] = 14   z[2,2,3] = 16   z[2,3,3] = 18

z[1,1,4] = 19   z[1,2,4] = 21   z[1,3,4] = 23
z[2,1,4] = 20   z[2,2,4] = 22   z[2,3,4] = 24
```

6.5. Matrix- and Vector-Valued Variables

The dump format for matrices and vectors, including arrays of matrices and vectors, is the same as that for arrays of the same shape.

Vector Dump Format

The following three declarations have the same dump format for their data.

```
real a[K];
vector[K] b;
row_vector[K] c;
```

Matrix Dump Format

The following declarations have the same dump format.

```
real a[M,N];
matrix[M,N] b;
```

Arrays of Vectors and Matrices

The key to understanding arrays is that the array indexing comes before any of the container indexing. That is, an array of vectors is just that — provide an index and get a vector. See Section Section 21.5 for more information on indexing and assignment.

For the dump data format, the following declarations have the same arrangement.

```
real a[M,N];
matrix[M,N] b;
vector[N] c[M];
row_vector[N] d[M];
```

Similarly, the following also have the same dump format.

```
real a[P,M,N];
matrix[M,N] b[P];
vector[N] c[P,M];
row_vector[N] d[P,M];
```

6.6. Integer- and Real-Valued Variables

There is no declaration in a dump file that distinguishes integer versus continuous values. If a value in a dump file's definition of a variable contains a decimal point (e.g., 132.3) or uses scientific notation (e.g., 1.323e2), Stan assumes that the values are real.

For a single value, if there is no decimal point, it may be assigned to an `int` or `real` variable in Stan. An array value may only be assigned to an `int` array if there is no decimal point or scientific notation in any of the values. This convention is compatible with the way R writes data.

The following dump file declares an integer value for y.

```
y <-
2
```

This definition can be used for a Stan variable y declared as `real` or as `int`. Assigning an integer value to a real variable automatically promotes the integer value to a real value.

Integer values may optionally be followed by L or l, denoting long integer values. The following example, where the type is explicit, is equivalent to the above.

```
y <-
2L
```

The following dump file provides a real value for y.

```
y <-
2.0
```

Even though this is a round value, the occurrence of the decimal point in the value, 2.0, causes Stan to infer that y is real valued. This dump file may only be used for variables y declared as real in Stan.

Scientific Notation

Numbers written in scientific notation may only be used for real values in Stan. R will write out the integer one million as 1e+06.

Infinite and Not-a-Number Values

Stan's reader supports infinite and not-a-number values for scalar quantities (see Section 21.2 for more information). Both infinite and not-a-number values are supported by Stan's dump-format readers.

Value	Preferred Form	Alternative Forms
positive infinity	Inf	Infinity, infinity
negative infinity	-Inf	-Infinity, -infinity
not a number	NaN	

These strings are not case sensitive, so `inf` may also be used for positive infinity, or `NAN` for not-a-number.

6.7. Quoted Variable Names

In order to support JAGS data files, variables may be double quoted. For instance, the following definition is legal in a dump file.

```
"y" <-
c(1,2,3)
```

6.8. Line Breaks

The line breaks in a dump file are required to be consistent with the way R reads in data. Both of the following declarations are legal.

```
y <- 2
y <-
3
```

Also following R, breaking before the assignment arrow are not allowed, so the following is invalid.

```
y
<- 2  # Syntax Error
```

Lines may also be broken in the middle of sequences declared using the `c(...)` notation., as well as between the comma following a sequence definition and the dimensionality declaration. For example, the following declaration of a $2 \times 2 \times 3$ array is valid.

```
y <-
structure(c(1,2,3,
4,5,6,7,8,9,10,11,
12), .Dim = c(2,2,
3))
```

Because there are no decimal points in the values, the resulting dump file may be
used for three-dimensional array variables declared as int or real.

6.9.   BNF Grammar for Dump Data

A more precise definition of the dump data format is provided by the following
(mildly templated) Backus-Naur form grammar.

```
definitions ::= definition+

definition ::= name ("<-" | '=') value optional_semicolon

name ::= char*
       | ''' char* '''
       | '"' char* '"'

value ::= value<int> | value<double>

value<T> ::= T
           | seq<T>
           | 'structure' '(' seq<T> ',' ".Dim" '=' seq<int> ')'

seq<int> ::= int ':' int
           | cseq<int>

seq<real> ::= cseq<real>

cseq<T> ::= 'c' '(' vseq<T> ')'

vseq<T> ::= T
          | T ',' vseq<T>
```

The template parameters T will be set to either int or real. Because Stan allows
promotion of integer values to real values, an integer sequence specification in the
dump data format may be assigned to either an integer- or real-based variable in
Stan.

Part III

Programming Techniques

7. Model Building as Software Development

Developing a Stan model is a software development process. Developing software is hard. Very hard. So many things can go wrong because there are so many moving parts and combinations of parts.

Software development practices are designed to mitigate the problems caused by the inherent complexity of software development. Unfortunately, many methodologies veer off into dogma, bean counting, or both. A couple we can recommend that provide solid, practical advice for developers are (Hunt and Thomas, 1999) and (McConnell, 2004). This section tries to summarize some of their advice.

7.1. Use Version Control

Version control software, such as Subversion or Git, should be in place before starting to code.[1] It may seem like a big investment to learn version control, but it's well worth it to be able to type a single command to revert to a previously working version or to get the difference between the current version and an old version. It's even better when you need to share work with others, even on a paper.

7.2. Make it Reproducible

Rather than entering commands on the command-line when running models (or entering commands directly into an interactive programming language like R or Python), try writing scripts to run the data through the models and produce whatever posterior analysis you need. Scripts can be written for the shell, R, or Python. Whatever language a script is in, it should be self contained and not depend on global variables having been set, other data being read in, etc.

Scripts are Good Documentation

It may seem like overkill if running the project is only a single line of code, but the script provides not only a way to run the code, but also a form of concrete documentation for what is run.

---

[1]Stan started using Subversion (SVN), then switched to the much more feature-rich Git package. Git does everything SVN does and a whole lot more. The price is a steeper learning curve. For individual or very-small-team development, SVN is just fine.

Randomization and Saving Seeds

Randomness defeats reproducibility. MCMC methods are conceptually randomized. Stan's samplers involve random initializations as well as randomization during each iteration (e.g., Hamiltonian Monte Carlo generates a random momentum in each iteration).

Computers are deterministic. There is no real randomness, just pseudo-random number generators. These operate by generating a sequence of random numbers based on a "seed." Stan (and other languages like R) can use time-based methods to generate a seed based on the time and date, or seeds can be provided to Stan (or R) in the form of long integers. Stan writes out the seed used to generate the data as well as the version number of the Stan software so that results can be reproduced at a later date.[2]

7.3. Make it Readable

Treating programs and scripts like other forms of writing for an audience provides an important perspective on how the code will be used. Not only might others want to read a program or model, the developer will want to read it later. One of the motivations of Stan's design was to make models self-documenting in terms of variable usage (e.g., data versus parameter), types (e.g., covariance matrix vs. unconstrained matrix) and sizes.

A large part of readability is consistency. Particularly in naming and layout. Not only of programs themselves, but the directories and files in which they're stored.

Readability of code is not just about comments (see Section Section 7.8 for commenting recommendations and syntax in Stan).

It is surprising how often the solution to a debugging or design problem occurs when trying to explain enough about the problem to someone else to get help. This can be on a mailing list, but it works best person-to-person. Finding the solution to your own problem when explaining it to someone else happens so frquently in software development that the listener is called a "rubber ducky," because they only have to nod along.[3]

---

[2]This also requires fixing compilers and hardware, because floating-point arithmetic does not have an absolutely fixed behavior across platforms or compilers, just operating parameters.

[3]Research has shown an actual rubber ducky won't work. For some reason, the rubber ducky must actually be capable of understanding the explanation.

7.4. Explore the Data

Although this should go without saying, don't just fit data blindly. Look at the data you actually have to understand its properties. If you're doing a logistic regression, is it separable? If you're building a multilevel model, do the basic outcomes vary by level? If you're fitting a linear regression, see whether such a model makes sense by scatterplotting x vs. y.

7.5. Design Top-Down, Code Bottom-Up

Software projects are almost always designed top-down from one or more intended use cases. Good software coding, on the other hand, is typically done bottom-up.

The motivation for top-down design is obvious. The motivation for bottom-up development is that it is much easier to develop software using components that have been thoroughly tested. Although Stan has no built-in support for either modularity or testing, many of the same principles apply.

The way the developers of Stan themselves build models is to start as simply as possibly, then build up. This is true even if we have a complicated model in mind as the end goal, and even if we have a very good idea of the model we eventually want to fit. Rather than building a hierarchical model with multiple interactions, covariance priors, or other complicated structure, start simple. Build just a simple regression with fixed (and fairly tight) priors. Then add interactions or additional levels. One at a time. Make sure that these do the right thing. Then expand.

7.6. Fit Simulated Data

One of the best ways to make sure your model is doing the right thing computationally is to generate simulated (i.e., "fake") data with known parameter values, then see if the model can recover these parameters from the data. If not, there is very little hope that it will do the right thing with data from the wild.

There are fancier ways to do this, where you can do things like run χ^2 tests on marginal statistics or follow the paradigm introduced in (Cook et al., 2006), which involves interval tests.

7.7. Debug by Print

Although Stan does not have a stepwise debugger or any unit testing framework in place, it does support the time-honored tradition of debug-by-printf. [4]

Stan supports print statements with one or more string or expression arguments. Because Stan is an imperative language, variables can have different values at different points in the execution of a program. Print statements can be invaluable for debugging, especially for a language like Stan with no stepwise debugger.

For instance, to print the value of variables y and z, use the following statement.

```
print("y=", y, " z=", z);
```

This print statement prints the string "y=" followed by the value of y, followed by the string " z=" (with the leading space), followed by the value of the variable z.

Each print statement is followed by a new line. The specific ASCII character(s) generated to create a new line are platform specific.

Arbitrary expressions can be used. For example, the statement

```
print("1+1=", 1+1);
```

will print "1 + 1 = 2" followed by a new line.

Print statements may be used anywhere other statements may be used, but their behavior in terms of frequency depends on how often the block they are in is evaluated. See Section 23.8 for more information on the syntax and evaluation of print statements.

7.8. Comments

Code Never Lies

The machine does what the code says, not what the documentation says. Docuementation, on the other hand, might not match the code. Code documentation easily rots as the code evolves if the documentation is not well maintained.

Thus it is always preferable to write readable code as opposed to documenting unreadable code. Every time you write a piece of documentation, ask yourself if there's a way to write the code in such a way as to make the documentation unnecessary.

---

[4]The "f" is not a typo — it's a historical artifact of the name of the `printf` function used for formatted printing in C.

Comment Styles in Stan

Stan supports C++-style comments; see Section 24.1 for full details. The recommended style is to use line-based comments for short comments on the code or to comment out one or more lines of code. Bracketed comments are then reserved for long documentation comments. The reason for this convention is that bracketed comments cannot be wrapped inside of bracketed comments.

What Not to Comment

When commenting code, it is usually safe to assume that you are writing the comments for other programmers who understand the basics of the programming language in use. In other words, don't comment the obvious. For instance, there is no need to have comments such as the following, which add nothing to the code.

```
y ~ normal(0,1);  // y has a unit normal distribution
```

A Jacobian adjustment for a hand-coded transform might be worth commenting, as in the following example.

```
exp(y) ~ normal(0,1);
// adjust for change of vars: y = log | d/dy exp(y) |
increment_log_prob(y);
```

It's an art form to empathize with a future code reader and decide what they will or won't know (or remember) about statistics and Stan.

What to Comment

It can help to document variable declarations if variables are given generic names like N, mu, and sigma. For example, some data variable declarations in an item-response model might be usefully commented as follows.

```
int<lower=1> N;   // number of observations
int<lower=1> I;   // number of students
int<lower=1> J;   // number of test questions
```

The alternative is to use longer names that do not require comments.

```
int<lower=1> n_obs;
int<lower=1> n_students;
int<lower=1> n_questions;
```

Both styles are reasonable and which one to adopt is mostly a matter of taste (mostly because sometimes models come with their own naming conventions which should be followed so as not to confuse readers of the code familiar with the statistical conventions).

Some code authors like big blocks of comments at the top explaining the purpose of the model, who wrote it, copyright and licensing information, and so on. The following bracketed comment is an example of a conventional style for large comment blocks.

```
/*
 * Item-Response Theory PL3 Model
 * -------------------------------------------------
 * Copyright: Joe Schmoe  <joe@schmoe.com>
 * Date:  19 September 2012
 * License: GPLv3
 */

data {
  ...
```

The use of leading asterisks helps readers understand the scope of the comment. The problem with including dates or other volatile information in comments is that they can easily get out of synch with the reality of the code. A misleading comment or one that is wrong is worse than no comment at all!

8. Containers: Arrays, Vectors, and Matrices

Stan provides three types of container objects: arrays, vectors, and matrices. The three types are not interchangeable. Vectors, matrices, and arrays are not assignable to one another, even if their dimensions are identical. A 3×4 matrix is a different kind of object in Stan than a 3×4 array.

8.1. Vectors and Matrices

Vectors and matrices are more limited kinds of data structures than arrays. Vectors are intrinsically one-dimensional collections of reals, whereas matrices are intrinsically two dimensional.

The intention of using matrix types is to call out their usage in the code. There are three situations in Stan where *only* vectors and matrices may be used,

- matrix arithmetic operations (e.g., matrix multiplication)
- linear algebra functions (e.g., eigenvalues and determinants), and
- multivariate function parameters and outcomes (e.g., multivariate normal distribution arguments).

Vectors and matrices cannot be typed to return integer values. They are restricted to `real` values.[1]

8.2. Arrays

Arrays, on the other hand, are intrinsically one-dimensional collections of other kinds of objects. The values in an array can be any type, so that arrays may contain values that are simple reals or integers, vectors, matrices, or other arrays. Arrays are the only way to store sequences of integers, and some functions in Stan, such as discrete distributions, require integer arguments.

A two-dimensional array is just an array of arrays, both conceptually and in terms of current implementation. When an index is supplied to an array, it returns the value at that index. When more than one index is supplied, this idexing operation is chained. For example, if `a` is a two-dimensional array, then `a[m,n]` is just a convenient shorthand for `a[m][n]`.

---

[1]This may change if Stan is called upon to do complicated integer matrix operations or boolean matrix operations. Integers are not appropriate inputs for linear algebra functions.

8.3. Efficiency Considerations

One of the motivations for Stan's underlying design is efficiency.

The underlying matrix and linear algebra operations are implemented in terms of data types from the Eigen C++ library. By having vectors and matrices as basic types, no conversion is necessary when invoking matrix operations or calling linear algebra functions.

Arrays, on the other hand, are implemented as instances of the C++ `std::vector` class (not to be confused with Eigen's `Eigen::Vector` class or Stan vectors). By implementing arrays this way, indexing is very efficient because values can be returned by reference rather than copied by value.

Matrices vs. Two-Dimensional Arrays

In Stan models, there are a few minor efficiency considerations in deciding between a two-dimensional array and a matrix, which may seem interchangeable at first glance.

First, matrices use a bit less memory than two-dimensional arrays. This is because they don't store a sequence of arrays, but just the data and the two dimensions.

Second, matrices store their data in row-major order. Furthermore, all of the data in a matrix is guaranteed to be contiguous in memory. This is an important consideration for optimized code because bringing in data from memory to cache is much more expensive than performing arithmetic operations with contemporary CPUs. Arrays, on the other hand, only guarantee that the values of primitive types are contiguous in memory; otherwise, they hold copies of their values (which are returned by reference wherever possible).

Third, both data structures are best traversed in the order in which they are stored. This also helps with memory locality. This is column-major for matrices, so the following order is appropriate.

```
matrix[M,N] a;
...
for (n in 1:N)
  for (m in 1:M)
    ... do something with a[m,n] ...
```

Arrays, on the other hand, should be traversed in row-major order.

```
real a[M,N];
...
for (m in 1:M)
  for (n in 1:N)
    ... do something with a[m,n] ...
```

The first use of `a[m,n]` should bring `a[m]` into memory. Overall, traversing matrices is more efficient than traversing arrays.

If `a` is a matrix, the notation `a[m]` picks out row `m` of that matrix. This is a rather inefficient operation for matrices. If indexing of vectors is needed, it is much better to declare an array of vectors. That is, this

```
row_vector[N] b[M];
...
for (m in 1:M)
    ... do something with row vector b[m] ...
```

is much more efficient than the pure matrix version

```
matrix b[M,N];
...
for (m in 1:M)
    ... do something with row vector b[m] ...
```

Similarly, indexing an array of column vectors is more efficient than using the `col` function to pick out a column of a matrix.

In contrast, whatever can be done as pure matrix algebra will be the fastest. So if I want to create a row of predictor-coefficient dot-products, it's more efficient to do this

```
matrix[N,K] x;    // predictors (aka covariates)
...
vector[K] beta;   // coeffs
...
vector[N] y_hat;  // linear prediction
...
y_hat <- x * beta;
```

than it is to do this

```
row_vector[K] x[N];    // predictors (aka covariates)
...
vector[K] beta;   // coeffs
...
vector[N] y_hat;  // linear prediction
...
for (n in 1:N)
  y_hat[n] <- x[n] * beta;
```

(Row) Vectors vs. One-Dimensional Arrays

For use purely as a container, there is really nothing to decide among vectors, row vectors and one-dimensional arrays. The `Eigen::Vector` template specialization and the `std::vector` template class are implemented very similarly as containers of `double` values (the type `real` in Stan). Only arrays in Stan are allowed to store integer values.

9. Missing Data & Partially Known Parameters

BUGS and R support mixed arrays of known and missing data. In BUGS, known and unknown values may be mixed as long as every unknown variable appears on the left-hand side of either an assignment or sampling statement.

9.1. Missing Data

Stan treats variables declared in the `data` and `transformed data` blocks as known and the variables in the `parameters` block as unknown.

The next section shows how to create a mixed array of known and unknown values as in BUGS. The recommended approach to missing data in Stan is slightly different than in BUGS. An example involving missing normal observations[1] could be coded as follows.

```
data {
  int<lower=0> N_obs;
  int<lower=0> N_miss;
  real y_obs[N_obs];
}
parameters {
  real mu;
  real<lower=0> sigma;
  real y_miss[N_miss];
}
model {
  for (n in 1:N_obs)
    y_obs[n] ~ normal(mu,sigma);
  for (n in 1:N_miss)
    y_miss[n] ~ normal(mu,sigma);
}
```

The number of observed and missing data points are coded as data with non-negative integer variables `N_obs` and `N_miss`. The observed data is provided as an array data variable `y_obs`. The missing data is coded as an array parameter, `y_miss`. The ordinary parameters being estimated, the location `mu` and scale `sigma`, are also coded as parameters. A better way to write the model would be to vectorize, so the body would be

---

[1] A more meaningful estimation example would involve a regression of the observed and missing observations using predictors that were known for each and specified in the `data` block.

```
    y_obs ~ normal(mu,sigma);
    y_miss ~ normal(mu,sigma);
```

The model contains one loop over the observed data and one over the missing data. This slight redundancy in specification leads to much more efficient sampling for missing data problems in Stan than the more general technique described in the next section.

9.2. Partially Known Parameters

In some situations, such as when a multivariate probability function has partially observed outcomes or parameters, it will be necessary to create a vector mixing known (data) and unknown (parameter) values. This can be done in Stan by creating a vector or array in the `transformed parameters` block and assigning to it.

The following example involves a bivariate covariance matrix in which the variances are known, but the covariance is not.

```
data {
  int<lower=0> N;
  vector[2] y[N];
  real<lower=0> var1;      real<lower=0> var2;
}
transformed data {
  real<upper=0> min_cov;
  real<lower=0> max_cov;
  max_cov <- sqrt(var1 * var2);
  min_cov <- -max_cov;
}
parameters {
  vector[2] mu;
  real<lower=min_cov,upper=max_cov> cov;
}
transformed parameters {
  matrix[2,2] sigma;
  sigma[1,1] <- var1;      sigma[1,2] <- cov;
  sigma[2,1] <- cov;       sigma[2,2] <- var2;
}
model {
 for (n in 1:N)
    y[n] ~ multi_normal(mu,sigma);
}
```

The variances are defined as data in variables `var1` and `var2`, whereas the covariance is defined as a parameter in variable `cov`. The 2×2 covariance matrix `sigma` is defined

as a transformed parameter, with the variances assigned to the two diagonal elements and the covariance to the two off-diagonal elements.

The constraint on the covariance declaration ensures that the resulting covariance matrix `sigma` is positive definite. The bound, plus or minus the square root of the product of the variances, is defined as transformed data so that it is only calculated once.

9.3. Efficiency Note

The missing-data example in the first section could be programmed with a mixed data and parameter array following the approach of the partially known parameter example in the second section. The behavior will be correct, but the computation is wasteful. Each parameter, be it declared in the `parameters` or `transformed parameters` block, uses an algorithmic differentiation variable which is more expensive in terms of memory and gradient-calculation time than a simple data variable. Furthermore, the copy takes up extra space and extra time.

10.   Truncated or Censored Data

Data in which measurements have been truncated or censored can be coded in Stan following their respective probability models.

10.1.   Truncated Distributions

Truncation in Stan is restricted to univariate distributions for which the corresponding log cumulative distribution function (cdf) and log completmentary cumulative distribution (ccdf) functions are available. See the subsection on truncated distributions in Section 23.3 for more information on truncated distributions, cdfs, and ccdfs.

10.2.   Truncated Data

Truncated data is data for which measurements are only reported if they fall above a lower bound, below an upper bound, or between a lower and upper bound.

Truncated data may be modeled in Stan using truncated distributions. For example, suppose the truncated data is y_n with an upper truncation point of $U = 300$ so that $y_n < 300$. In Stan, this data can be modeled as following a truncated normal distribution for the observations as follows.

```
data {
  int<lower=0> N;
  real U;
  real<upper=U> y[N];
}
parameters {
  real mu;
  real<lower=0> sigma;
}
model {
  for (n in 1:N)
    y[n] ~ normal(mu,sigma) T[,U];
}
```

The model declares an upper bound U as data and constrains the data for y to respect the constraint; this will be checked when the data is loaded into the model before sampling begins.

This model implicitly uses an improper flat prior on the scale and location parameters; these could be given priors in the model using sampling statements.

Constraints and Out-of-Bounds Returns

If the sampled variate in a truncated distribution lies outside of the truncation range, the probability is zero, so the log probability will evaluate to $-\infty$. For instance, if variate y is sampled with the statement.

```
for (n in 1:N)
  y[n] ~ normal(mu,sigma) T[L,U];
```

then if the value of y[n] is less than the value of L or greater than the value of U, the sampling statement produces a zero-probability estimate.

To avoid variables straying outside of truncation bounds, appropriate constraints are required. For example, if y is a parameter in the above model, the declaration should constrain it to fall between the values of L and U.

```
parameters {
  real<lower=L,upper=U> y[N];
  ...
```

If in the above model, L or U is a parameter and y is data, then L and U must be appropriately constrained so that all data is in range and the value of L is less than that of U (if they are equal, the parameter range collapses to a single point and the Hamiltonian dynamics used by the sampler break down). The following declarations ensure the bounds are well behaved.

```
parameters {
  real<upper=min(y)> L; // L < y[n]
  real<lower=fmax(L,max(y))> U; // L < U; y[n] < U
```

Note that for pairs of real numbers, the function fmax is used rather than max.

Unknown Truncation Points

If the truncation points are unknown, they may be estimated as parameters. This can be done with a slight rearrangement of the variable declarations from the model in the previous section with known truncation points.

```
data {
  int<lower=1> N;
  real y[N];
}
parameters {
  real<upper = min(y)> L;
  real<lower = max(y)> U;
  real mu;
```

```
    real<lower=0> sigma;
}
model {
  L ~ ...;
  U ~ ...;
  for (n in 1:N)
    y[n] ~ normal(mu,sigma) T[L,U];
}
```

Here there is a lower truncation point L which is declared to be less than or equal
to the minimum value of y. The upper truncation point U is declared to be larger
than the maximum value of y. This declaration, although dependent on the data,
only enforces the constraint that the data fall within the truncation bounds. With N
declared as type int<lower=1>, there must be at least one data point. The constraint
that L is less than U is enforced indirectly, based on the non-empty data.

    The ellipses where the priors for the bounds L and U should go should be filled
in with a an informative prior in order for this model to not concentrate L strongly
around min(y) and U strongly around max(y).

10.3. Censored Data

Censoring hides values from points that are too large, too small, or both. Unlike with
truncated data, the number of data points that were censored is known. The textbook
example is the household scale which does not report values above 300 pounds.

Estimating Censored Values

One way to model censored data is to treat the censored data as missing data that
is constrained to fall in the censored range of values. Because Stan does not allow
unknown values in its arrays or matrices, the censored values must be represented
explicitly.

```
data {
  int<lower=0> N_obs;
  int<lower=0> N_cens;
  real<lower=0> y_obs[N_obs];
  real<lower=max(y_obs)> U;
}
parameters {
  real<lower=U> y_cens[N_cens];
  real mu;
  real<lower=0> sigma;
```

```
}
model {
  for (n in 1:N_obs)
    y_obs[n] ~ normal(mu,sigma);
  for (n in 1:N_cens)
    y_cens[n] ~ normal(mu,sigma);
}
```

Because the censored data array `y_cens` is declared to be a parameter, it will be sampled along with the location and scale parameters `mu` and `sigma`. Because the censored data array `y_cens` is declared to have values of type `real<lower=U>`, all imputed values for censored data will be greater than U. The imputed censored data affects the location and scale parameters through the last sampling statement in the model.

Integrating out Censored Values

Although it is wrong to ignore the censored values in estimating location and scale, it is not necessary to impute values. Instead, the values can be integrated out. Each censored data point has a probability of

$$\Pr[y > U] = \int_U^\infty \mathsf{Normal}(y|\mu,\sigma)\,dy = 1 - \Phi\left(\frac{y-\mu}{\sigma}\right),$$

where $\Phi()$ is the unit normal cumulative distribution function. With M censored observations, the total probability on the log scale is

$$\log \prod_{m=1}^{M} \Pr[y_m > U] = \log\left(1 - \Phi\left(\frac{y-\mu}{\sigma}\right)\right)^M = M \log\left(1 - \Phi\left(\frac{y-\mu}{\sigma}\right)\right)$$

Although Stan implements Φ with the function `Phi`, Stan also provides the cumulative distribution function `normal_cdf`, defined by

$$\mathsf{normal\_cdf}(y,\mu,\sigma) = \Phi\left(\frac{y-\mu}{\sigma}\right).$$

The following model assumes that the censoring point is known, so it is declared as data.

```
data {
  int<lower=0> N_obs;
  int<lower=0> N_cens;
  real<lower=0> y_obs[N_obs];
  real<lower=max(y_obs)> U;
}
```

```
parameters {
  real mu;
  real<lower=0> sigma;
}
model {
  for (n in 1:N_obs)
    y_obs[n] ~ normal(mu,sigma);
  increment_log_prob(N_cens * log1m(normal_cdf(U,mu,sigma)));
}
```

For the observed values in `y_obs`, the normal sampling model is used without trunca-
tion. The log probability is directly incremented using the calculated log cumulative
normal probability of the censored data items. The built-in function `log1m` is used,
which maps x to $\log(1 - x)$ in an arithmetically stable way.

To deal with situations where the censoring point variable `U` is unknown, the dec-
laration of `U` should be moved from the data block to the parameters block.

11.   Mixture Modeling

Mixture models of an outcome assume that the outcome is drawn from one of several distributions, the identity of which is controlled by a categorical mixing distribution. Mixture models typically have multimodal densities with modes near the modes of the mixture components. Mixture models may be parameterized in several ways, as described in the following sections.

11.1.   Latent Discrete Parameterization

One way to parameterize a mixture model is with a latent categorical variable indicating which mixture component was responsible for the outcome. For example, consider K normal distributions with locations $\mu_k \in \mathbb{R}$ and scales $\sigma_k \in (0, \infty)$. Now consider mixing them in proportion θ, where $\theta_k \geq 0$ and $\sum_{k=1}^{K} \theta_k = 1$ (i.e., θ lies in the unit K-simplex). For each outcome y_n there is a latent variable z_n in $\{1, \dots, K\}$ with a categorical distribution parameterized by θ,

$$z_n \sim \mathsf{Categorical}(\theta).$$

The variable y_n is distributed according to the parameters of the mixture component z_n,

$$y_n \sim \mathsf{Normal}(\mu_{z[n]}, \sigma_{z[n]}).$$

This model is not directly supported by Stan because it involves discrete parameters z_n, but Stan can sample μ and σ by summing out the z parameter as described in the next section.

11.2.   Summing out the Responsibility Parameter

To implement the normal mixture model outlined in the previous section in Stan, the discrete parameters can be summed out of the model. If Y is a mixture of K normal distributions with locations μ_k and scales σ_k with mixing proportions θ in the unit K-simplex, then

$$p_Y(y) = \sum_{k=1}^{K} \theta_k \, \mathsf{Normal}(\mu_k, \sigma_k).$$

For example, the mixture of $\mathsf{Normal}(-1, 2)$ and $\mathsf{Normal}(3, 1)$ with mixing proportion $\theta = (0.3, 0.7)^\top$ can be implemented in Stan as follows.

```
parameters {
  real y;
```

```
    }
    model {
      increment_log_prob(log_sum_exp(log(0.3)
                                       + normal_log(y,-1,2),
                                     log(0.7)
                                       + normal_log(y,3,1)));
    }
```

The log probability term is derived by taking

$$
\begin{aligned}
\log p_Y(y) &= \log\left(0.3 \times \mathsf{Normal}(y|-1,2) + 0.7 \times \mathsf{Normal}(y|3,1)\right) \\
&= \log\left(\exp(\log(0.3 \times \mathsf{Normal}(y|-1,2))) \right. \\
&\qquad \left. + \exp(\log(0.7 \times \mathsf{Normal}(y|3,1)))\right) \\
&= \log\_sum\_exp\left(\log(0.3) + \log \mathsf{Normal}(y|-1,2),\right. \\
&\qquad \left. \log(0.7) + \log \mathsf{Normal}(y|3,1)\right).
\end{aligned}
$$

Given the scheme for representing mixtures, it may be moved to an estimation setting, where the locations, scales, and mixture components are unknown. Further generalizing to a number of mixture components specified as data yields the following model.

```
    data {
      int<lower=1> K;          // number of mixture components
      int<lower=1> N;          // number of data points
      real y[N];               // observations
    }
    parameters {
      simplex[K] theta;        // mixing proportions
      real mu[K];              // locations of mixture components
      real<lower=0,upper=10> sigma[K];  // scales of mixture components
    }
    model {
      real ps[K];              // temp for log component densities
      for (k in 1:K) {
        mu[k] ~ normal(0,10);
      }
      for (n in 1:N) {
        for (k in 1:K) {
          ps[k] <- log(theta[k])
                   + normal_log(y[n],mu[k],sigma[k]);
        }
        increment_log_prob(log_sum_exp(ps));
      }
    }
```

The model involves K mixture components and N data points. The mixing proportion parameter theta is declared to be a unit *K*-simplex, whereas the component location parameter mu and scale parameter sigma are both defined to be arrays of size K. The values in the scale array sigma are constrained to be non-negative, and have an upper bound of 10. Since no prior is explicitly defined for the sigma parameters, their implicit prior distributions are uniform over their ranges. The model declares a local array variable ps to be size K and uses it to accumulate the contributions from the mixture components.

The locations and scales are drawn from simple priors for the sake of this example, but could be anything supported by Stan. The mixture components could even be modeled hierarchically.

The main action is in the loop over data points n. For each such point, the log of $\theta_k \times \text{Normal}(y_n | \mu_k, \sigma_k)$ is calculated and added to the array ps. Then the log probability is incremented with the log sum of exponentials of those values.

12.   Regression Models

Stan supports regression models from simple linear regressions to multilevel gener-
alized linear models. Coding regression models in Stan is very much like coding them
in BUGS.

12.1.   Linear Regression

The simplest linear regression model is the following, with a single predictor and a
slope and intercept coefficient, and normally distributed noise. This model can be
written using standard regression notation as

$$Y_n = \alpha + \beta x_n + \epsilon_n \quad \text{where} \quad \epsilon_n \sim \mathsf{Normal}(0, \sigma).$$

This is equivalent to the following sampling involving the residual,

$$Y_n - (\alpha + \beta X_n) \sim \mathsf{Normal}(0, \sigma),$$

and reducing still further, to

$$Y_n \sim \mathsf{Normal}(\alpha + \beta X_n, \sigma).$$

This latter form of the model is coded in Stan as follows.

```
data {
  int<lower=0> N;
  vector[N] x;
  vector[N] y;
}
parameters {
  real alpha;
  real beta;
  real<lower=0> sigma;
}
model {
  for (n in 1:N)
    y[n] ~ normal(alpha + beta * x[n], sigma);
}
```

There are N observations, each with predictor x[n] and outcome y[n]. The intercept
and slope parameters are alpha and beta. The model assumes a normally distributed
noise term with scale sigma. This model has improper priors for the two regression
coefficients.

Matrix Notation and Vectorization

The sampling statement in the previous model can be vectorized and written equivalently as follows:

```
model {
  y ~ normal(alpha + beta * x, sigma);
}
```

The main difference is that the vectorized form is much faster.[1]

In general, Stan allows the arguments to distributions such as `normal` to be vectors. If any of the other arguments are vectors or arrays, they have to be the same size. If any of the other arguments is a scalar, it is reused for each vector entry. See Chapter 26 for more information on vectorization.

The other reason this works is that Stan's arithmetic operators are overloaded to perform matrix arithmetic on matrices. In this case, because `x` is of type `vector` and `beta` of type `real`, the expression `beta * x` is of type `vector`. Because Stan supports vectorization, a regression model with more than one predictor can be written directly using matrix notation.

```
data {
  int<lower=0> N;   // number of data items
  int<lower=0> K;   // number of predictors
  matrix[N,K] x;    // predictor matrix
  vector[N] y;      // outcome vector
}
parameters {
  real alpha;           // intercept
  vector[N] beta;       // coefficients for predictors
  real<lower=0,upper=10> sigma;  // error scale
}
model {
  y ~ normal(x * beta, sigma);  // likelihood
}
```

The constraint on `sigma` gives it a uniform prior on $(0, 10)$. The sampling statement in the model above is equivalent to

---

[1]Unlike in Python and R, which are interpreted, Stan is translated to C++ and compiled, so loops and assignment statements are fast. Vectorized code is faster in Stan because (a) the expression tree used to compute derivatives can be simplified, leading to fewer virtual function calls, and (b) computations that would be repeated in the looping version, such as `log(sigma)` in the above model, will be computed once and reused.

```
    for (n in 1:N)
      y ~ normal(x[n] * beta, sigma);
```

With Stan's matrix indexing scheme, `x[n]` picks out row `n` of the matrix `x`; because `beta` is a column vector, the product `x[n] * beta` is a scalar of type `real`.

12.2. Coefficient and Noise Priors

There are several ways in which the model in the previous section can be generalized. For example, weak priors can be assigned to the coefficients as follows.

```
    alpha ~ normal(0,100);
    beta ~ normal(0,100);
```

And an upper bound to `sigma` can be given in order to implicitly give it a uniform prior.

```
    real<lower=0,upper=100> sigma;
```

More informative priors based the (half) Cauchy distribution are coded as follows.

```
    alpha ~ cauchy(0,2.5);
    beta ~ cauchy(0,2.5);
    sigma ~ cauchy(0,2.5);
```

The regression coefficients `alpha` and `beta` are unconstrained, but `sigma` must be positive and properly requires the half-Cauchy distribution. Although Stan supports truncated distributions with half distributions being a special case, it is not necessary here because the full distribution is proportional when the parameters are constant.[2]

12.3. Robust Noise Models

The standard approach to linear regression is to model the noise term ϵ as having a normal distribution. From Stan's perspective, there is nothing special about normally distributed noise. For instance, robust regression can be accommodated by giving the noise term a Student-t distribution. To code this in Stan, the sampling distribution is changed to the following.

---

[2]Stan does not (yet) support truncated Cauchy distributions. The distributions which may be truncated are listed for discrete distributions in Part VI and for continuous distributions in Part VII. Available truncated distributions may be found in the index by looking for suffix `_cdf`.

```
data {
  ...
  real<lower=0> nu;
}
...
model {
  for (n in 1:N)
    y[n] ~ student_t(nu, alpha + beta * x[n], sigma);
}
```

The degrees of freedom constant `nu` is specified as data.

12.4.   Logistic and Probit Regression

For binary outcomes, either of the closely related logistic or probit regression models
may be used. These generalized linear models vary only in the link function they
use to map linear predictions in $(-\infty, \infty)$ to probability values in $(0, 1)$. Their respec-
tive link functions, the logistic function and the unit normal cumulative distribution
function, are both sigmoid functions (i.e., they are both *S*-shaped).

   A logistic regression model with one predictor and an intercept is coded as fol-
lows.

```
data {
  int<lower=0> N;
  real x[N];
  int<lower=0,upper=1> y[N];
}
parameters {
  real alpha;
  real beta;
}
model {
  for (n in 1:N)
    y[n] ~ bernoulli(inv_logit(alpha + beta * x[n]));
}
```

The noise parameter is built into the Bernoulli formulation here rather than specified
directly.

   Logistic regression is a kind of generalized linear model with binary outcomes
and the log odds (logit) link function. The inverse of the link function appears in the
model.

Other link functions may be used in the same way. For example, probit regression uses the cumulative normal distribution function, which is typically written as

$$\Phi(x) = \int_{-\infty}^{x} \mathsf{Normal}(y|0,1)\,dy.$$

The cumulative unit normal distribution function Φ is implemented in Stan as the function `Phi`. The probit regression model may be coded in Stan by replacing the logistic model's sampling statement with the following.

```
y[n] ~ bernoulli(Phi(alpha + beta * x[n]));
```

A fast approximation to the cumulative unit normal distribution function Φ is implemented in Stan as the function `Phi_approx`. The approximate probit regression model may be coded with the following.

```
y[n] ~ bernoulli(Phi_approx(alpha + beta * x[n]));
```

12.5. Multi-Logit Regression

Multiple outcome forms of logistic regression can be coded directly in Stan. For instance, suppose there are K possible outcomes for each output variable y_n. Also suppose that there is a D-dimensional vector x_n of predictors for y_n. The multi-logit model with $\mathsf{Normal}(0,5)$ priors on the coefficients is coded as follows.

```
data {
  int K;
  int N;
  int D;
  int y[N];
  vector[D] x[N];
}
parameters {
  matrix[K,D] beta;
}
model {
  for (k in 1:K)
    for (d in 1:D)
      beta[k,d] ~ normal(0,5);
  for (n in 1:N)
    y[n] ~ categorical(softmax(beta * x[n]));
}
```

The softmax function is defined for a K-vector $y \in \mathbb{R}^K$ by

$$\text{softmax}(y) = \left(\frac{\exp(y_1)}{\sum_{k=1}^{K} \exp(y_k)}, \ldots, \frac{\exp(y_K)}{\sum_{k=1}^{K} \exp(y_k)} \right).$$

The result is in the unit K-simplex and thus appropriate to use as the parameter for a categorical distribution.

Constraints on Data Declarations

The data block in the above model is defined without constraints on sizes K, N, and D or on the outcome array y. Constraints on data declarations provide error checking at the point data is read (or transformed data is defined), which is before sampling begins. Constraints on data declarations also make the model author's intentions more explicit, which can help with readability. The above model's declarations could be tightened to

```
int<lower=2> K;
int<lower=0> N;
int<lower=1> D;
int<lower=1,upper=K> y[N];
```

These constraints arise because the number of categories, K, must be at least two in order for a categorical model to be useful. The number of data items, N, can be zero, but not negative; unlike R, Stan's for-loops always move forward, so that a loop extent of 1:N when N is equal to zero ensures the loop's body will not be executed. The number of predictors, D, must be at least one in order for beta * x[n] to produce an appropriate agrument for softmax(). The categorical outcomes y[n] must be between 1 and K in order for the discrete sampling to be well defined.

Constraints on data declarations are optional. Constraints on parameters declared in the parameters block, on the other hand, are *not* optional—they are required to ensure support for all parameter values satisfying their constraints. Constraints on transformed data, transformed parameters, and generated quantities are also optional.

Identifiability

Because softmax is invariant under adding a constant to each component of its input, the model is typically only identified if there is a suitable prior on the coefficients.

An alternative is to use $K - 1$ vectors by fixing one of them to be zero. See Section 9.2 for an example of how to mix known quantities and unknown quantities in a vector.

12.6. Ordered Logistic and Probit Regression

Ordered regression for an outcome $y_n \in \{1, \ldots, K\}$ with predictors $x_n \in \mathbb{R}^D$ is determined by a single coefficient vector $\beta \in \mathbb{R}^D$ along with a sequence of cutpoints $c \in \mathbb{R}^{D-1}$ sorted so that $c_d < c_{d+1}$. The discrete output is k if the linear predictor $x_n \beta$ falls between c_{k-1} and c_k, assuming $c_0 = -\infty$ and $c_K = \infty$. The noise term is fixed by the form of regression, with examples for ordered logistic and ordered probit models.

Ordered Logistic Regression

The ordered logistic model can be coded in Stan using the `ordered` data type for the cutpoints and the built-in `ordered_logistic` distribution.

```
data {
  int<lower=2> K;
  int<lower=0> N;
  int<lower=1> D;
  int<lower=1,upper=K> y[N];
  row_vector[D] x[N];
}
parameters {
  vector[D] beta;
  ordered[K-1] c;
}
model {
  for (n in 1:N)
    y[n] ~ ordered_logistic(x[n] * beta, c);
}
```

The vector of cutpoints `c` is declared as `ordered[K-1]`, which guarantees that `c[k]` is less than `c[k+1]`.

   If the cutpoints were assigned independent priors, the constraint effectively truncates the joint prior to support over points that satisfy the ordering constraint. Luckily, Stan does not need to compute the effect of the constraint on the normalizing term because the probability is needed only up to a proportion.

Ordered Probit

An ordered probit model could be coded in a manner similar to the BUGS encoding of an ordered logistic model.

```
data {
  int<lower=2> K;
```

```
    int<lower=0> N;
    int<lower=1> D;
    int<lower=1,upper=K> y[N];
    row_vector[D] x[N];
  }
  parameters {
    vector[D] beta;
    ordered[K-1] c;
  }
  model {
    vector[K] theta;
    for (n in 1:N) {
      real eta;
      eta <- x[n] * beta;
      theta[1] <- 1 - Phi(eta - c[1]);
      for (k in 2:(K-1))
        theta[k] <- Phi(eta - c[k-1]) - Phi(eta - c[k]);
      theta[K] <- Phi(eta - c[K-1]);
      y[n] ~ categorical(theta);
    }
  }
```

The logistic model could also be coded this way by replacing Phi with inv_logit, though the built-in encoding based on the softmax transform is more efficient and more numerically stable. A small efficiency gain could be achieved by computing the values Phi(eta - c[k]) once and storing them for re-use.

12.7. Hierarchical Logistic Regression

The simplest multilevel model is a hierarchical model in which the data is grouped into L distinct categories (or levels). An extreme approach would be to completely pool all the data and estimate a common vector of regression coefficients β. At the other extreme, an approach would no pooling assigns each level l its own coefficient vector β_l that is estimated separately from the other levels. A hierarchical model is an intermediate solution where the degree of pooling is determined by the data and a prior on the amount of pooling.

Suppose each binary outcome $y_n \in \{0, 1\}$ has an associated level, $ll_n \in \{1, \ldots, L\}$. Each outcome will also have an associated predictor vector $x_n \in \mathbb{R}^D$. Each level l gets its own coefficient vector $\beta_l \in \mathbb{R}^D$. The hierarchical structure involves drawing the coefficients $\beta_{l,d} \in \mathbb{R}$ from a prior that is also estimated with the data. This hierarchically estimated prior determines the amount of pooling. If the data in each level are very

similar, strong pooling will be reflected in low hierarchical variance. If the data in the levels are dissimilar, weaker pooling will be reflected in higher hierarchical variance.

The following model encodes a hierarchical logistic regression model with a hierarchical prior on the regression coefficients.

```
data {
  int<lower=1> D;
  int<lower=0> N;
  int<lower=1> L;
  int<lower=0,upper=1> y[N];
  int<lower=1,upper=L> ll[N];
  row_vector[D] x[N];
}
parameters {
  real mu[D];
  real<lower=0,upper=1000> sigma[D];
  vector[D] beta[L];
}
model {
  for (d in 1:D) {
    mu[d] ~ normal(0,100);
    for (l in 1:L)
      beta[l,d] ~ normal(mu[d],sigma[d]);
  }
  for (n in 1:N)
    y[n] ~ bernoulli(inv_logit(x[n] * beta[ll[n]]));
}
```

12.8. Item-Response Theory Models

Item-response theory (IRT) models the situation in which a number of students each answer one or more of a group of test questions. The model is based on parameters for the ability of the students, the difficulty of the questions, and in more articulated models, the discriminativeness of the questions and the probability of guessing correctly; see (Gelman and Hill, 2007, pps. 314–320) for a textbook introduction to hierarchical IRT models and (Curtis, 2010) for encodings of a range of IRT models in BUGS.

Data Declaration with Missingness

The data provided for an IRT model may be declared as follows to account for the fact that not every student is required to answer every question.

```
data {
  int<lower=1> J;              // number of students
  int<lower=1> K;              // number of questions
  int<lower=1> N;              // number of observations
  int<lower=1,upper=J> jj[N];  // student for observation n
  int<lower=1,upper=K> kk[N];  // question for observation n
  int<lower=0,upper=1> y[N];   // correctness for observation n
}
```

This declares a total of N student-question pairs in the data set, where each n in 1:N indexes a binary observation y[n] of the correctness of the answer of student jj[n] on question kk[n].

The prior hyperparameters will be hard coded in the rest of this section for simplicity, though they could be coded as data in Stan for more flexibility.

1PL (Rasch) Model

The 1PL item-response model, also known as the Rasch model, has one parameter (1P) for questions and uses the logistic link function (L). This model is distributed with Stan in the file src/models/misc/irt/irt.stan.

The model parameters are declared as follows.

```
parameters {
  real delta;        // mean student ability
  real alpha[J];     // ability of student j - mean ability
  real beta[K];      // difficulty of question k
}
```

The parameter alpha[j] is the ability coefficient for student j and beta[k] is the difficulty coefficient for question k. The non-standard parameterization used here also includes an intercept term delta, which represents the average student's response to the average question.[3] The model itself is as follows.

```
model {
  alpha ~ normal(0,1);      // informative true prior
  beta ~ normal(0,1);       // informative true prior
  delta ~ normal(.75,1);    // informative true prior
  for (n in 1:N)
    y[n] ~ bernoulli_logit(alpha[jj[n]] - beta[kk[n]] + delta);
}
```

---

[3](Gelman and Hill, 2007) treat the δ term equivalently as the location parameter in the distribution of student abilities.

This model uses the logit-parameterized Bernoulli distribution, where

$$\mathtt{bernoulli\_logit}(y|\alpha) = \mathtt{bernoulli}(y|\mathrm{logit}^{-1}(\alpha)).$$

The key to understanding it is the term inside the `bernoulli_logit` distribution, from which it follows that

$$\Pr[Y_n = 1] = \mathrm{logit}^{-1}(\alpha_{jj[n]} - \beta_{kk[n]} + \delta).$$

The model suffers from additive identifiability issues without the priors. For example, adding a term ξ to each α_j and β_k results in the same predictions. The use of priors for α and β located at 0 identifies the parameters; see (Gelman and Hill, 2007) for a discussion of identifiability issues and alternative approaches to identification.

For testing purposes, the IRT 1PL model distributed with Stan uses informative priors that match the actual data generation process used to simulate the data in R (the simulation code is supplied in the same directory as the models). This is unrealistic for most practical applications, but allows Stan's inferences to be validated. A simple sensitivity analysis with fatter priors shows that the posterior is fairly sensitive to the prior even with 400 students and 100 questions and only 25% missingness at random. For real applications, the priors should be fit hierarchically along with the other parameters, as described in the next section.

Multilevel 2PL Model

The simple 1PL model described in the previous section is generalized in this section with the addition of a discrimination parameter to model how noisy a question is and by adding multilevel priors for the student and question parameters.

The model parameters are declared as follows.

```
parameters {
  real delta;                 // mean student ability
  real alpha[J];              // ability for j - mean
  real beta[K];               // difficulty for k
  real log_gamma[K];          // discrim for k
  real<lower=0> sigma_alpha;  // sd of abilities
  real<lower=0> sigma_beta;   // sd of difficulties
  real<lower=0> sigma_gamma;  // sd of log discrim
}
```

The parameters should be clearer after the model definition.

```
model {
  alpha ~ normal(0,sigma_alpha);
  beta ~ normal(0,sigma_beta);
```

```
    log_gamma ~ normal(0,sigma_gamma);
    delta ~ cauchy(0,5);
    sigma_alpha ~ cauchy(0,5);
    sigma_beta ~ cauchy(0,5);
    sigma_gamma ~ cauchy(0,5);
    for (n in 1:N)
      y[n] ~ bernoulli_logit(
                exp(log_gamma[kk[n]])
                * (alpha[jj[n]] - beta[kk[n]] + delta) );
  }
```

First, the predictor inside the `bernoulli_logit` term is equivalent to the predictor of the 1PL model multiplied by the discriminativeness for the question, `exp(log_gamma[kk[n]])`. The parameter `log_gamma[k]` represents how discriminative a question is, with log discriminations above 0 being less (because their exponentiation drives the predictor away from zero, which drives the prediction away from 0.5) and discriminations below 0 being more noisy (driving the predictor toward zero and hence the prediction toward 0.5).

The intercept term `delta` can't be modeled hierarchically, so it is given a weakly informative $\text{Cauchy}(0, 5)$ prior. Similarly, the scale terms, `sigma_alpha`, `sigma_beta`, and `sigma_gamma`, are given half-Cauchy priors. The truncation in the half-Cauchy prior is implicit; explicit truncation is not necessary because the log probability need only be calculated up to a proportion and the scale variables are constrained to $(0, \infty)$ by their declarations.

13.   Time-Series Models

Times series data come arranged in temporal order. This chapter presents two kinds of time series models, regression-like models such as autogression and moving average models, and hidden Markov models.

13.1.   Autoregressive Models

A first-order autoregressive model (AR(1)) with normal noise takes each point y_n in a sequence y to be generated according to

$$y_n \sim \text{Normal}(\alpha + \beta y_{n-1}, \sigma).$$

That is, the expected value of y_n is $\alpha + \beta y_{n-1}$, with noise scaled as σ.

AR(1) Models

With improper flat priors on the regression coefficients for slope (β), intercept (α), and noise scale (σ), the Stan program for the AR(1) model is as follows.

```
data {
  int<lower=0> N;
  real y[N];
}
parameters {
  real alpha;
  real beta;
  real sigma;
}
model {
  for (n in 2:N)
    y[n] ~ normal(alpha + beta*y[n-1], sigma);
}
```

The first observed data point, y[1], is not modeled here.

Extensions to the AR(1) Model

Proper priors of a range of different families may be added for the regression coefficients and noise scale. The normal noise model can be changed to a Student-t distribution or any other distribution with unbounded support. The model could also be made hierarchical if multiple series of observations are available.

To enforce the estimation of a stationary AR(1) process, the slope coefficient `beta` may be constrained with bounds as follows.

```
real<lower=-1,upper=1> beta;
```

In practice, such a constraint is not recommended. If the data is not stationary, it is best to discover this while fitting the model. Stationary parameter estimates can be encouraged with a prior favoring values of `beta` near zero.

AR(2) Models

Extending the order of the model is also straightforward. For example, an AR(2) model could be coded with the second-order coefficient `gamma` and the following model statement.

```
for (n in 3:N)
  y[n] ~ normal(alpha + beta*y[n-1] + gamma*y[n-2], sigma);
```

AR(K) Models

A general model where the order is itself given as data can be coded by putting the coefficients in an array and computing the linear predictor in a loop.

```
data {
  int<lower=0> K;
  int<lower=0> N;
  real y[N];
}
parameters {
  real alpha;
  real beta[K];
  real sigma;
}
model {
  for (n in (K+1):N) {
    real mu;
    mu <- alpha;
    for (k in 1:K)
      mu <- mu + beta[k] * y[n-k];
    y[n] ~ normal(mu, sigma);
  }
}
```

ARCH(1) Models

Econometric and financial time-series models usually assume heteroscedasticity (i.e., they allow the scale of the noise terms defining the series to vary over time). The simplest such model is the autoregressive conditional heteroscedasticity (ARCH) model (Engle, 1982). Unlike the autoregressive model AR(1), which modeled the mean of the series as varying over time but left the noise term fixed, the ARCH(1) model takes the scale of the noise terms to vary over time but leaves the mean term fixed. Of course, models could be defined where both the mean and scale vary over time; the econometrics literature presents a wide range of time-series modeling choices.

The ARCH(1) model is typically presented as the following sequence of equations, where r_t is the observed return at time point t and μ, α_0, and α_1 are unknown regression coefficient parameters.

$$
\begin{aligned}
r_t &= \mu + a_t \\
a_t &= \sigma_t \epsilon_t \\
\epsilon_t &\sim \text{Normal}(0,1) \\
\sigma_t^2 &= \alpha_0 + \alpha_1 a_{t-1}^2
\end{aligned}
$$

In order to ensure the noise terms σ_t^2 are positive, the scale coefficients are constrained to be positive, $\alpha_0, \alpha_1 > 0$. To ensure stationarity of the time series, the slope is constrained to to be less than one, $\alpha_1 < 1$.[1] The ARCH(1) model may be coded directly in Stan as follows.

```
data {
  int<lower=0> T;   // number of time points
  real r[T];        // return at time t
}
parameters {
  real mu;                        // average return
  real<lower=0> alpha0;           // noise intercept
  real<lower=0,upper=1> alpha1;   // noise slope
}
model {
  for (t in 2:T)
    r[t] ~ normal(mu, sqrt(alpha0 + alpha1 * pow(r[t-1] - mu,2)));
}
```

The loop in the model is defined so that the return at time $t = 1$ is not modeled; the model in the next section shows how to model the return at $t = 1$. The model can be vectorized to be more efficient; the model in the next section provides an example.

---

[1] In practice, it can be useful to remove the constraint to test whether a non-stationary set of coefficients provides a better fit to the data.

13.2. Modeling Temporal Heteroscedasticity

A set of variables is homoscedastic if their variances are all the same; the variables are heteroscedastic if they do not all have the same variance. Heteroscedastic time-series models allow the noise term to vary over time.

GARCH(1,1) Models

The basic generalized autoregressive conditional heteroscedasticity (GARCH) model, GARCH(1,1), extends the ARCH(1) model by including the squared previous difference in return from the mean at time $t - 1$ as a predictor of volatility at time t, defining

$$\sigma_t^2 = \alpha_0 + \alpha_1 a_{t-1}^2 + \beta_1 \sigma_{t-1}^2.$$

To ensure the scale term is positive and the resulting time series stationary, the coefficients must all satisfy $\alpha_0, \alpha_1, \beta_1 > 0$ and the slopes $\alpha_1 + \beta_1 < 1$.

```
data {
  int<lower=0> T;
  real r[T];
  real<lower=0> sigma1;
}
parameters {
  real mu;
  real<lower=0> alpha0;
  real<lower=0,upper=1> alpha1;
  real<lower=0,upper=(1-alpha1)> beta1;
}
transformed parameters {
  real<lower=0> sigma[T];
  sigma[1] <- sigma1;
  for (t in 2:T)
    sigma[t] <- sqrt(alpha0
                     + alpha1 * pow(r[t-1] - mu, 2)
                     + beta1 * pow(sigma[t-1], 2));
}
model {
  r ~ normal(mu,sigma);
}
```

To get the recursive definition of the volatility regression off the ground, the data declaration includes a non-negative value `sigma1` for the scale of the noise at $t = 1$.

The constraints are coded directly on the parameter declarations. This declaration is order-specific in that the constraint on `beta1` depends on the value of `alpha1`.

A transformed parameter array of non-negative values `sigma` is used to store the scale values at each time point. The definition of these values in the transformed parameters block is where the regression is now defined. There is an intercept `alpha0`, a slope `alpha1` for the squared difference in return from the mean at the previous time, and a slope `beta1` for the previous noise scale squared. Finally, the whole regression is inside the `sqrt` function because Stan requires scale (deviation) parameters (not variance parameters) for the normal distribution.

With the regression in the transformed parameters block, the model reduces a single vectorized sampling statement. Because `r` and `sigma` are of length `T`, all of the data is modeled directly.

13.3.   Moving Average Models

A moving average model uses previous errors as predictors for future outcomes. For a moving average model of order Q, MA(Q), there is an overall mean parameter μ and regression coefficients θ_q for previous error terms. With ϵ_t being the noise at time t, the model for outcome y_t is defined by

$$y_t = \mu + \theta_1 \epsilon_{t-1} + \cdots + \theta_Q \epsilon_{t-Q} + \epsilon_t,$$

with the noise term ϵ_t for outcome y_t modeled as normal,

$$\epsilon_t \sim \mathsf{Normal}(0, \sigma).$$

In a proper Bayesian model, the parameters μ, θ, and σ must all be given priors.

MA(2) Example

An MA(2) model can be coded in Stan as follows.

```
data {
  int<lower=3> T;  // number of observations
  vector[T] y;     // observation at time T
}
parameters {
  real mu;                 // mean
  real<lower=0> sigma;  // error scale
  vector[2] theta;      // lag coefficients
}
transformed parameters {
  vector[T] epsilon;    // error terms
  epsilon[1] <- y[1] - mu;
  epsilon[2] <- y[2] - mu - theta[1] * epsilon[1];
```

```
    for (t in 3:T)
      epsilon[t] <- ( y[t] - mu
                      - theta[1] * epsilon[t - 1]
                      - theta[2] * epsilon[t - 2] );
  }
  model {
    mu ~ cauchy(0,2.5);
    theta ~ cauchy(0,2.5);
    sigma ~ cauchy(0,2.5);
    for (t in 3:T)
      y[t] ~ normal(mu
                    + theta[1] * epsilon[t - 1]
                    + theta[2] * epsilon[t - 2],
                    sigma);
  }
```

The error terms ϵ_t are defined as transformed parameters in terms of the observations and parameters. The definition of the sampling statement (defining the likelihood) follows the definition, which can only be applied to y_n for $n > Q$. In this example, the parameters are all given Cauchy (half-Cauchy for σ) priors, although other priors can be used just as easily.

This model could be improved in terms of speed by vectorizing the sampling statement in the model block. Vectorizing the calculation of the ϵ_t could also be sped up by using a dot product instead of a loop.

Vectorized MA(Q) Model

A general MA(Q) model with a vectorized sampling probability may be defined as follows.

```
    data {
      int<lower=0> Q;  // num previous noise terms
      int<lower=3> T;  // num observations
      vector[T] y;     // observation at time t
    }
    parameters {
      real mu;              // mean
      real<lower=0> sigma;  // error scale
      vector[Q] theta;      // error coeff, lag -t
    }
    transformed parameters {
      vector[T] epsilon;    // error term at time t
      for (t in 1:T) {
        epsilon[t] <- y[t] - mu;
```

```
      for (q in 1:min(t-1,Q))
        epsilon[t] <- epsilon[t] - theta[q] * epsilon[t - q];
    }
  }
model {
  vector[T] eta;
  mu ~ cauchy(0,2.5);
  theta ~ cauchy(0,2.5);
  sigma ~ cauchy(0,2.5);
  for (t in 1:T) {
    eta[t] <- mu;
    for (q in 1:min(t-1,Q))
      eta[t] <- eta[t] + theta[q] * epsilon[t - q];
  }
  y ~ normal(eta,sigma);
}
```

Here all of the data is modeled, with missing terms just dropped from the regressions as in the calculation of the error terms. Both models converge very quickly and mix very well at convergence, with the vectorized model being quite a bit faster (per iteration, not to converge — they compute the same model).

13.4. Autoregressive Moving Average Models

Autoregressive moving-average models (ARMA), combine the predictors of the autoregressive model and the oving average model. An ARMA(1,1) model, with a single state of history, can be encoded in Stan as follows.

```
data {
  int<lower=1> T;            // num observations
  real y[T];                 // observed outputs
}
parameters {
  real mu;                   // mean coeff
  real phi;                  // autoregression coeff
  real theta;                // moving avg coeff
  real<lower=0> sigma;       // noise scale
}
model {
  vector[T] nu;              // prediction for time t
  vector[T] err;             // error for time t
  nu[1] <- mu + phi * mu;    // assume err[0] == 0
  err[1] <- y[1] - nu[1];
```

```
  for (t in 2:T) {
    nu[t] <- mu + phi * y[t-1] + theta * err[t-1];
    err[t] <- y[t] - nu[t];
  }
  mu ~ normal(0,10);          // priors
  phi ~ normal(0,2);
  theta ~ normal(0,2);
  sigma ~ cauchy(0,5);
  err ~ normal(0,sigma);     // likelihood
}
```

The data is declared in the same way as the other time-series regressions. Here the are parameters for the mean output `mu` and error scale `sigma`, as well as regression coefficients `phi` for the autoregression and `theta` for the moving average component of the model.

In the model block, the local vector `nu` stores the predictions and `err` the errors. These are computed similarly to the errors in the moving average models described in the previous section.

The priors are weakly informative for stationary processes. The likelihood only involves the error term, which is efficiently vectorized here.

Often in models such as these, it is desirable to inspect the calculated error terms. This could easily be accomplished in Stan by declaring `err` as a transformed parameter, then defining it the same way as in the model above. The vector `nu` could still be a local variable, only now it will be in the transformed parameter block.

Wayne Folta suggested encoding the model without local vector variables as follows.

```
model {
  real err;
  mu ~ normal(0,10);
  phi ~ normal(0,2);
  theta ~ normal(0,2);
  sigma ~ cauchy(0,5);
  err <- y[1] - mu + phi * mu;
  err ~ normal(0,sigma);
  for (t in 2:T) {
    err <- y[t] - (mu + phi * y[t-1] + theta * err);
    err ~ normal(0,sigma);
  }
}
```

This approach to ARMA models provides a nice example of how local variables, such as `err` in this case, can be reused in Stan. Folta's approach could be extended to

higher order moving-average models by storing more than one error term as a local variable and reassigning them in the loop.

Both encodings are very fast. The original encoding has the advantage of vectorizing the normal distribution, but it uses a bit more memory. A halfway point would be to vectorize just `err`.

13.5.   Stochastic Volatility Models

Stochastic volatility models treat the volatility (i.e., variance) of a return on an asset, such as an option to buy a security, as following a latent stochastic process in discrete time (Kim et al., 1998). The data consist of mean corrected (i.e., centered) returns y_t on an underlying asset at T equally spaced time points. Kim et al. formulate a typical stochastic volatility model using the following regression-like equations, with a latent parameter h_t for the log volatility, along with parameters μ for the mean log volatility, and ϕ for the persistence of the volatility term. The variable ϵ_t represents the white-noise shock (i.e., multiplicative error) on the asset return at time t, whereas δ_t represents the shock on volatility at time t.

$$y_t = \epsilon_t \exp(h_t/2),$$

$$h_{t+1} = \mu + \phi(h_t - \mu) + \delta_t \sigma$$

$$h_1 \sim \text{Normal}\left(\mu, \frac{\sigma}{\sqrt{1 - \phi^2}}\right)$$

$$\epsilon_t \sim \text{Normal}(0, 1); \quad \delta_t \sim \text{Normal}(0, 1)$$

Rearranging the first line, $\epsilon_t = y_t \exp(-h_t/2)$, allowing the sampling distribution for y_t to be written as

$$y_t \sim \text{Normal}(0, \exp(h_t/2)).$$

The recurrence equation for h_{t+1} may be combined with the scaling and sampling of δ_t to yield the sampling distribution

$$h_t \sim \text{Normal}(\mu + \phi(h_t - \mu), \sigma).$$

This formulation can be directly encoded, as shown in the following Stan model, which is also available in the file `<stan>/src/models/misc/moving-avg/stochastic-volatility.stan` along with R code to simulate data from the model for testing.

```
data {
  int<lower=0> T;   // # time points (equally spaced)
```

```
  vector[T] y;        // mean corrected return at time t
}
parameters {
  real mu;                       // mean log volatility
  real<lower=-1,upper=1> phi;    // persistence of volatility
  real<lower=0> sigma;           // white noise shock scale
  vector[T] h;                   // log volatility at time t
}
model {
  phi ~ uniform(-1,1);
  sigma ~ cauchy(0,5);
  mu ~ cauchy(0,10);
  h[1] ~ normal(mu, sigma / sqrt(1 - phi * phi));
  for (t in 2:T)
    h[t] ~ normal(mu + phi * (h[t - 1] -  mu), sigma);
  for (t in 1:T)
    y[t] ~ normal(0, exp(h[t] / 2));
}
```

Compared to the Kim et al. formulation, the Stan model adds priors for the parameters ϕ, σ, and μ. Note that the shock terms ϵ_t and δ_t do not appear explicitly in the model, although they could be calculated efficiently in a generated quantities block.

The posterior of a stochastic volatility model such as this one typically has high posterior variance. For example, simulating 500 data points from the above model with $\mu = -1.02$, $\phi = 0.95$, and $\sigma = 0.25$ leads to 95% posterior intervals for μ of $(-1.23, -0.54)$, for ϕ of $(0.82, 0.98)$ and for σ of $(0.16, 0.38)$.

The samples using NUTS show a high degree of autocorrelation among the samples, both for this model and the stochastic volatility model evaluated in (Hoffman and Gelman, 2011, 2013). Using a non-diagonal mass matrix provides faster convergence and more effective samples than a diagonal mass matrix, but will not scale to large values of T.

It is relatively straightforward to speed up the effective samples per second generated by this model by one or more orders of magnitude. First, the sampling statements for return y is easily vectorized to

```
y ~ normal(0, exp(h / 2));
```

This speeds up the iterations, but does not change the effective sample size because the underlying parameterization and log probability function have not changed. Mixing is improved by by reparameterizing in terms of a standardized volatility, then rescaling. This requires a standardized parameter h_std to be declared instead of h.

```
parameters {
  ...
```

```
    vector[T] h_std;                  // std log volatility time t
```

The original value of h is then defined in a transformed parameter block.

```
transformed parameters {
  vector[T] h;                    // log volatility at time t
  h <- h_std * sigma;
  h[1] <- h[1] / sqrt(1 - phi * phi);
  h <- h + mu;
  for (t in 2:T)
    h[t] <- h[t] + phi * (h[t-1] - mu);
}
```

Finally, the sampling statement for h[1] and loop for sampling h[2] to h[T] are replaced with a single vectorized unit normal sampling statement.

```
model {
  ...
  h_std ~ normal(0,1);
```

Although the original model can take hundreds and sometimes thousands of iterations to converge, the reparameterized model reliably converges in tens of iterations. Mixing is also dramatically improved, which results in higher effective sample sizes per iteration. Finally, each iteration runs in roughly a quarter of the time of the original iterations.

13.6. Hidden Markov Models

A hidden Markov model (HMM) generates a sequence of T output variables y_t conditioned on a parallel sequence of latent categorical state variables $z_t \in \{1, \ldots, K\}$. These "hidden" state variables are assumed to form a Markov chain so that z_t is conditionally independent of other variables given z_{t-1}. This Markov chain is parameterized by a transition matrix θ where θ_k is a K-simplex for $k \in \{1, \ldots, K\}$. The probability of transitioning to state z_t from state z_{t-1} is

$$z_t \sim \mathsf{Categorical}(\theta_{z[t-1]}).$$

The output y_t at time t is generated conditionally independently based on the latent state z_t. This section describes HMMs with a simple categorical model for outputs $y_t \in \{1, \ldots, V\}$. The categorical distribution for latent state k is parameterized by a V-simplex ϕ_k. The observed output y_t at time t is generated based on the hidden state indicator z_t at time t,

$$y_t \sim \mathsf{Categorical}(\phi_{z[t]}).$$

In short, HMMs form a discrete mixture model where the mixture component indicators form a latent Markov chain.

Supervised Parameter Estimation

In the situation where the hidden states are known, the following naive model can be used to fit the parameters θ and ϕ. (This model is distributed with Stan on the path `<stan>/src/models/misc/hmm/hmm.stan`.)

```
data {
  int<lower=1> K;  // num categories
  int<lower=1> V;  // num words
  int<lower=0> T;  // num instances
  int<lower=1,upper=V> w[T]; // words
  int<lower=1,upper=K> z[T]; // categories
  vector<lower=0>[K] alpha;  // transit prior
  vector<lower=0>[V] beta;   // emit prior
}
parameters {
  simplex[K] theta[K];  // transit probs
  simplex[V] phi[K];    // emit probs
}
model {
  for (k in 1:K)
    theta[k] ~ dirichlet(alpha);
  for (k in 1:K)
    phi[k] ~ dirichlet(beta);
  for (t in 1:T)
    w[t] ~ categorical(phi[z[t]]);
  for (t in 2:T)
    z[t] ~ categorical(theta[z[t - 1]]);
}
```

Explicit Dirichlet priors have been provided for θ_k and ϕ_k; dropping these two statements would implicitly take the prior to be uniform over all valid simplexes.

Start-State and End-State Probabilities

Although workable, the above description of HMMs is incomplete because the start state z_1 is not modeled (the index runs from 2 to T). If the data are conceived as a subsequence of a long-running process, the probability of z_1 should be set to the stationary state probabilities in the Markov chain. In this case, there is no distinct end to the data, so there is no need to model the probability that the sequence ends at z_T.

An alternative conception of HMMs is as models of finite-length sequences. For example, human language sentences have distinct starting distributions (usually a capital letter) and ending distributions (usually some kind of punctuation). The simplest way to model the sequence boundaries is to add a new latent state $K+1$, generate the first state from a categorical distribution with parameter vector θ_{K+1}, and restrict the transitions so that a transition to state $K + 1$ is forced to occur at the end of the sentence and is prohibited elsewhere.

Calculating Sufficient Statistics

The naive HMM estimation model presented above can be sped up dramatically by replacing the loops over categorical distributions with a single multinomial distribution. A complete implementation is available in the Stan source distribution at path `<stan>/src/models/misc/hmm/hmm-sufficient.stan`. The data is declared as before, but now a transformed data blocks computes the sufficient statistics for estimating the transition and emission matrices.

```
transformed data {
  int<lower=0> trans[K,K];
  int<lower=0> emit[K,V];
  for (k1 in 1:K)
    for (k2 in 1:K)
      trans[k1,k2] <- 0;
  for (t in 2:T)
    trans[z[t - 1], z[t]] <- 1 + trans[z[t - 1], z[t]];
  for (k in 1:K)
    for (v in 1:V)
      emit[k,v] <- 0;
  for (t in 1:T)
    emit[z[t], w[t]] <- 1 + emit[z[t], w[t]];
}
```

The likelihood component of the model based on looping over the input is replaced with multinomials as follows.

```
model {
  ...
  for (k in 1:K)
    trans[k] ~ multinomial(theta[k]);
  for (k in 1:K)
    emit[k] ~ multinomial(phi[k]);
}
```

In a continuous HMM with normal emission probabilities could be sped up in the same way by computing sufficient statistics.

Analytic Posterior

With the Dirichlet-multinomial HMM, the posterior can be computed analytically because the Dirichlet is the conjugate prior to the multinomial. The following example, available in `<stan>/src/models/hmm/hmm-analytic.stan`, illustrates how a Stan model can define the posterior analytically. This is possible in the Stan language because the model only needs to define the conditional probability of the parameters given the data up to a proportion, which can be done by defining the (unnormalized) joint probability or the (unnormalized) conditional posterior, or anything in between.

The model has the same data and parameters as the previous models, but now computes the posterior Dirichlet parameters in the transformed data block.

```
transformed data {
  vector<lower=0>[K] alpha_post[K];
  vector<lower=0>[V] beta_post[K];
  for (k in 1:K)
    alpha_post[k] <- alpha;
  for (t in 2:T)
    alpha_post[z[t-1],z[t]] <- alpha_post[z[t-1],z[t]] + 1;
  for (k in 1:K)
    beta_post[k] <- beta;
  for (t in 1:T)
    beta_post[z[t],w[t]] <- beta_post[z[t],w[t]] + 1;
}
```

The posterior can now be written analytically as follows.

```
model {
  for (k in 1:K)
    theta[k] ~ dirichlet(alpha_post[k]);
  for (k in 1:K)
    phi[k] ~ dirichlet(beta_post[k]);
}
```

Semisupervised Estimation

HMMs can be estimated in a fully unsupervised fashion without any data for which latent states are known. The resulting posteriors are typically extremely multimodal. An intermediate solution is to use semisupervised estimation, which is based on a combination of supervised and unsupervised data. Implementing this estimation

strategy in Stan requires calculating the probability of an output sequence with an unknown state sequence. This is a marginalization problem, and for HMMs, it is computed with the so-called forward algorithm.

In Stan, the forward algorithm is coded as follows (the full model is in `<stan>` `/src/models/misc/hmm/hmm-semisup.stan`). First, two additional data variable are declared for the unsupervised data.

```
data {
  ...
  int<lower=1> T_unsup;  // num unsupervised items
  int<lower=1,upper=V> u[T_unsup]; // unsup words
  ...
```

The model for the supervised data does not change; the unsupervised data is handled with the following Stan implementation of the forward algorithm.

```
model {
  ...
  {
    real acc[K];
    real gamma[T_unsup,K];
    for (k in 1:K)
      gamma[1,k] <- log(phi[k,u[1]]);
    for (t in 2:T_unsup) {
      for (k in 1:K) {
        for (j in 1:K)
          acc[j] <- gamma[t-1,j] + log(theta[j,k]) + log(phi[k,u[t]]);
        gamma[t,k] <- log_sum_exp(acc);
      }
    }
    increment_log_prob(log_sum_exp(gamma[T_unsup]));
  }
```

The forward values `gamma[t,k]` are defined to be the log marginal probability of the inputs `u[1],...,u[t]` up to time `t` and the latent state being equal to `k` at time `t`; the previous latent states are marginalized out. The first row of `gamma` is initialized by setting `gamma[1,k]` equal to the log probability of latent state `k` generating the first output `u[1]`; as before, the probability of the first latent state is not itself modeled. For each subsequent time `t` and output `j`, the value `acc[j]` is set to the probability of the latent state at time `t-1` being `j`, plus the log transition probability from state `j` at time `t-1` to state `k` at time `t`, plus the log probability of the output `u[t]` being generated by state `k`. The `log_sum_exp` operation just multiplies the probabilities for each prior state `j` on the log scale in an arithmetically stable way.

The brackets provide the scope for the local variables `acc` and `gamma`; these could have been declared earlier, but it is clearer to keep their declaration near their use.

Predictive Inference

Given the transition and emission parameters, $\theta_{k,k'}$ and $\phi_{k,v}$ and an observation sequence $u_1,\ldots,u_T \in \{1,\ldots,V\}$, the Viterbi (dynamic programming) algorithm computes the state sequence which is most likely to have generated the observed output u.

The Viterbi algorithm can be coded in Stan in the generated quantities block as follows. The predictions here is the most likely state sequence `y_star[1]`, ..., `y_star[T_unsup]` underlying the array of observations `u[1]`, ..., `u[T_unsup]`. Because this sequence is determined from the transition probabilities `theta` and emission probabilities `phi`, it may be different from sample to sample in the posterior.

```
generated quantities {
  int<lower=1,upper=K> y_star[T_unsup];
  real log_p_y_star;
  {
    int back_ptr[T_unsup,K];
    real best_logp[T_unsup,K];
    real best_total_logp;
    for (k in 1:K)
      best_logp[1,K] <- log(phi[k,u[1]]);
    for (t in 2:T_unsup) {
      for (k in 1:K) {
        best_logp[t,k] <- negative_infinity();
        for (j in 1:K) {
          real logp;
          logp <- best_logp[t-1,j]
                    + log(theta[j,k]) + log(phi[k,u[t]]);
          if (logp > best_logp[t,k]) {
            back_ptr[t,k] <- j;
            best_logp[t,k] <- logp;
          }
        }
      }
    }
    log_p_y_star <- max(best_logp[T_unsup]);
    for (k in 1:K)
      if (best_logp[T_unsup,k] == log_p_y_star)
        y_star[T_unsup] <- k;
```

```
      for (t in 1:(T_unsup - 1))
        y_star[T_unsup - t] <- back_ptr[T_unsup - t + 1,
                                        y_star[T_unsup - t + 1]];
    }
  }
```

The bracketed block is used to make the three variables `back_ptr`, `best_logp`, and `best_total_logp` local so they will not be output. The variable `y_star` will hold the label sequence with the highest probability given the input sequence `u`. Unlike the forward algorithm, where the intermediate quantities were total probability, here they consist of the maximum probabilty `best_logp[t,k]` for the sequence up to time `t` with final output category `k` for time `t`, along with a backpointer to the source of the link. Following the backpointers from the best final log probability for the final time `t` yields the optimal state sequence.

This inference can be run for the same unsupervised outputs `u` as are used to fit the semisupervised model. The above code can be found in the same model file as the unsupervised fit. This is the Bayesian approach to inference, where the data being reasoned about is used in a semisupervised way to train the model. It is not "cheating" because the underlying states for `u` are never observed — they are just estimated along with all of the other parameters.

If the outputs `u` are not used for semisupervised estimation but simply as the basis for prediction, the result is equivalent to what is represented in the BUGS modeling language via the cut operation. That is, the model is fit independenlty of `u`, then those parameters used to find the most likely state to have generated `u`.

14. Measurement Error and Meta-Analysis

Most quantities used in statistical models arise from measurements. Most of these measurements are taken with some error. When the measurement error is small relative to the quantity being measured, its effect on a model are usually small. When measurement error is large relative to the quantity being measured, or when very precise relations can be estimated being measured quantities, it is useful to introduce an explicit model of measurement error.

14.1. Bayesian Measurement Error Model

A Bayesian approach to measurement error can be formulated directly by treating the true quantities being measured as missing data (Clayton, 1992; Richardson and Gilks, 1993). This requires a model of how the measurements are derived from the true values.

Regression with Measurement Error

Before considering regression with measurement error, first consider a linear regression model where the observed data for N cases includes a predictor x_n and outcome y_n. In Stan, a linear regression for y based on x with a slope and intercept is modeled as follows.

```
data {
  int<lower=0> N;        // number of cases
  real x[N];             // predictor (covariate)
  real y[N];             // outcome (variate)
}
parameters {
  real alpha;            // intercept
  real beta;             // slope
  real<lower=0> sigma;   // outcome noise
}
model {
  y ~ normal(alpha + beta * x, sigma);
  alpha ~ normal(0,10);
  beta ~ normal(0,10);
  sigma ~ cauchy(0,5);
}
```

Now suppose that the true values of the predictors x_n are not known, but for each n, a measurement x_n^{meas} of x_n is available. If the error in measurement can be

modeled, the measured value x_n^{meas} can be modeled in terms of the true value x_n plus measurement noise. The true value x_n is treated as missing data and estimated along with other quantities in the model. A very simple approach is to assume the measurement error is normal with known deviation τ. This leads to the following regression model with constant measurement error.

```
data {
  ...
  real x_meas[N];     // measurement of x
  real<lower=0> tau;  // measurement noise
}
parameters {
  real x[N];          // unknown true value
  ...
}
model {
  x_meas ~ normal(x, tau);   // measurement model
  y ~ normal(alpha + beta * x, sigma);
  ...
}
```

The regression coefficients `alpha` and `beta` and regression noise scale `sigma` are the same as before, but now `x` is declared as a parameter rather than as data. The data is now `x_meas`, which is a measurement of the true `x` value with noise scale `tau`. The model then specifies that the measurement error for `x_meas[n]` given true value `x[n]` is normal with deviation `tau`.

A simple generalization of the above model is to allow the measurement noise term `tau` to vary with item. This only requires changing its declaration in the data block to

```
real<lower=0> tau[N];  // measurement noise for case n
```

In cases where the measurement errors are not normal, richer measurement error models may be specified.

Modeling the True Values

Although no prior is specified for the true value `x`, the posterior will be proper for the above model because

$$\text{Normal}(x|\mu, \Sigma) = \text{Normal}(\mu|x, \Sigma).$$

Nevertheless, it is common to provide some model of the true value x in terms of other covariates. For instance, (Clayton, 1992) introduces an exposure model for the

unknown (but noisily measured) risk factors x in terms of known (without measurement error) risk factors c. A simple model would regress x_n on the covariates c_n with noise term υ,

$$x_n \sim \mathsf{Normal}(\gamma^\top c, \upsilon).$$

This can be coded in Stan just like any other regression. And, of course, other exposure models can be provided.

14.2.   Meta-Analysis

Meta-analysis aims to pool the data from several studies, such as the application of a tutoring program in several schools or treatment using a drug in several clinical trials.

   The Bayesian framework is particularly convenient for meta-analysis, because each previous study can be treated as providing a noisy measurement of some underlying quantity of interest. The model then follows directly from two components, a prior on the underlying quantities of interest and a measurement-error style model for each of the studies being analyzed.

Treatment Effects in Controlled Studies

Suppose the data in question arise from a total of M studies providing paired binomial data for a treatment and control group. For instance, the data might be post-surgical pain reduction under a treatment of ibuprofen (Warn et al., 2002) or mortality after myocardial infarction under a treatment of beta blockers (Gelman et al., 2003, Section 5.6).

Data

The clinical data consists of J trials, each with n^t treatment cases, n^c control cases, r^t successful outcomes among those treated and r^c successful outcomes among those in the control group. This data can be declared in Stan as follows.[1]

```
data {
  int<lower=0> J;
  int<lower=0> n_t[J];  // num cases, treatment
  int<lower=0> r_t[J];  // num successes, treatment
  int<lower=0> n_c[J];  // num cases, control
  int<lower=0> r_c[J];  // num successes, control
}
```

---

[1]Stan's integer constraints are not powerful enough to express the constraint that r_t[j] ≤ n_t[j], but this constraint could be checked in the transformed data block.

Converting to Log Odds and Standard Error

Although the clinical trial data is binomial in its raw format, it may be transformed to an unbounded scale by considering the log odds ratio

$$
y_j = \log\left(\frac{r_j^t/(n_j^t - r_j^t)}{r_j^c/(n_j^c - r_j^c)}\right) \;=\; \log\left(\frac{r_j^t}{n_j^t - r_j^t}\right) - \log\left(\frac{r_j^c}{n_j^c - r_j^c}\right)
$$

and corresponding standard errors

$$
\sigma_j = \sqrt{\frac{1}{r_i^T} + \frac{1}{n_i^T - r_i^T} + \frac{1}{r_i^C} + \frac{1}{n_i^C - r_i^C}}.
$$

The log odds and standard errors can be defined in a transformed parameter block, though care must be taken not to use integer division (see Section 28.1).

```
transformed data {
  real y[J];
  real<lower=0> sigma[J];
  for (j in 1:J)
    y[j] <- log(r_t[j]) - log(n_t[j] - r_t[j])
            - (log(r_c[j]) - log(n_c[j] - r_c[j]));
  for (j in 1:J)
    sigma[j] <- sqrt(1.0/r_t[i] + 1.0/(n_t[i] - r_t[i])
                     + 1.0/r_c[i] + 1.0/(n_c[i] - r_c[i]));
}
```

This definition will be problematic if any of the success counts is zero or equal to the number of trials. If that arises, a direct binomial model will be required or other transforms must be used than the unregularized sample log odds.

Non-Hierarchical Model

With the transformed data in hand, two standard forms of meta-analysis can be applied. The first is a so-called "fixed effects" model, which assumes a single parameter for the global odds ratio. This model is coded in Stan as follows.

```
parameters {
  real theta;   // global treatment effect, log odds
}
model {
  y ~ normal(theta,sigma);
}
```

The sampling statement for y is vectorized; it has the same effect as the following.

```
    for (j in 1:J)
      y[j] ~ normal(theta,sigma[j]);
```

It is common to include a prior for `theta` in this model, but it is not strictly necessary for the model to be proper because y is fixed and $\mathsf{Normal}(y|\mu,\sigma) = \mathsf{Normal}(\mu|y,\sigma)$.

Hierarchical Model

To model so-called "random effects," where the treatment effect may vary by clinical trial, a hierarchical model can be used. The parameters include per-trial treatment effects and the hierarchical prior parameters, which will be estimated along with other unknown quantities.

```
parameters {
  real theta[J];      // per-trial treatment effect
  real mu;            // mean treatment effect
  real<lower=0> tau;  // deviation of treatment effects
}
model {
  y ~ normal(theta,sigma);
  theta ~ normal(mu,tau);
  mu ~ normal(0,10);
  tau ~ cauchy(0,5);
}
```

Although the vectorized sampling statement for y appears unchanged, the parameter `theta` is now a vector. The sampling statement for `theta` is also vectorized, with the hyperparameters `mu` and `tau` themselves being given wide priors compared to the scale of the data.

Rubin (1981) provided a hierarchical Bayesian meta-analysis of the treatment effect of Scholastic Aptitude Test (SAT) coaching in eight schools based on the sample treatment effect and standard error in each school. The model provided for this data in (Gelman et al., 2003, Section 5.5) is included with the data in the Stan distribution in directory `src/models/misc/eight-schools/`.

Extensions and Alternatives

Smith et al. (1995) and Gelman et al. (2003, Section 19.4) provide meta-analyses based directly on binomial data. Warn et al. (2002) consider the modeling implications of using alternatives to the log-odds ratio in transforming the binomial data.

If trial-specific predictors are available, these can be included directly in a regression model for the per-trial treatment effects θ_j.

15.   Clustering Models

Unsupervised methods for organizing data into groups are collectively referred to as clustering. This chapter describes the implementation in Stan of two widely used statistical clustering models, soft K-means and latent Dirichlet allocation (LDA). In addition, this chapter includes naive Bayesian classification, which can be viewed as a form of clustering which may be supervised. These models are typically expressed using discrete parameters for cluster assignments. Nevertheless, they can be implemented in Stan like any other mixture model by marginalizing out the discrete parameters (see Chapter 11).

15.1.   Soft K-Means

K-means clustering is a method of clustering data represented as D-dimensional vectors. Specifically, there will be N items to be clustered, each represented as a vector $y_n \in \mathbb{R}^D$. In the "soft" version of K-means, the assignments to clusters will be probabilistic.

Geometric Hard K-Means Clustering

K-means clustering is typically described geometrically in terms of the following algorithm, which assumes the number of clusters K and data vectors y as input.

1. For each n in $1 : N$, randomly assign vector y_n to a cluster in 1:K;

2. Repeat

   (a) For each cluster k in 1:K, compute the cluster centroid μ_k by averaging the vectors assigned to that cluster;

   (b) For each n in $1 : N$, reassign y_n to the cluster k to for which the (Euclidean) distance from y_n to μ_k is smallest;

   (c) If no vectors changed cluster, return the cluster assignments.

This algorithm is guaranteed to terminate.

Soft K-Means Clustering

Soft K-means clustering treats the cluster assignments as probability distributions over the clusters. Because of the connection between Euclidean distance and multivariate normal models with a fixed covariance, soft K-means can be expressed (and coded in Stan) as a multivariate normal mixture model.

In the full generative model, each data point n in 1:N is assigned a cluster $z_n \in 1{:}K$ with symmetric uniform probability,

$$z_n \sim \mathsf{Categorical}(\mathbf{1}/K),$$

where $\mathbf{1}$ is the unit vector of K dimensions, so that $\mathbf{1}/K$ is the symmetric K-simplex. Thus the model assumes that each data point is drawn from a hard decision about cluster membership. The softness arises only from the uncertainty about which cluster generated a data point.

The data points themselves are generated from a multivariate normal distribution whose parameters are determined by the cluster assignment z_n,

$$y_n \sim \mathsf{Normal}(\mu_{z[n]}, \Sigma_{z[n]})$$

The sample implementation in this section assumes a fixed unit covariance matrix shared by all clusters k,

$$\Sigma_k = \mathsf{diag\_matrix}(\mathbf{1}),$$

so that the log multivariate normal can be implemented directly up to a proportion by

$$\mathsf{Normal}\,(y_n | \mu_k, \mathsf{diag\_matrix}(\mathbf{1})) \propto \exp\left(-\frac{1}{2}\sum_{d=1}^{D}\left(\mu_{k,d} - y_{n,d}\right)^2\right).$$

The spatial perspective on K-means arises by noting that the inner term is just half the negative Euclidean distance from the cluster mean μ_k to the data point y_n.

Stan Implementation of Soft K-Means

The following model is available in the Stan distribution (along with an R program to randomly generate data sets and a sample data set) in the directory `stan/src/models/misc/soft-k-means`.

```
data {
  int<lower=0> N;  // number of data points
  int<lower=1> D;  // number of dimensions
  int<lower=1> K;  // number of clusters
  vector[D] y[N];  // observations
}
transformed data {
  real<upper=0> neg_log_K;
  neg_log_K <- -log(K);
}
parameters {
  vector[D] mu[K]; // cluster means
```

```
    }
    transformed parameters {
      real<upper=0> soft_z[N,K]; // log unnormalized clusters
      for (n in 1:N)
        for (k in 1:K)
          soft_z[n,k] <- neg_log_K
                         - 0.5 * dot_self(mu[k] - y[n]);
    }
    model {
      // prior
      for (k in 1:K)
        mu[k] ~ normal(0,1);

      // likelihood
      for (n in 1:N)
        increment_log_prob(log_sum_exp(soft_z[n]));
    }
```

There is an independent unit normal prior on the centroid parameters; this prior could be swapped with other priors, or even a hierarchical model to fit an overall problem scale and location.

The only parameter is mu, where mu[k] is the centroid for cluster k. The transformed parameters soft_z[n] contain the log of the unnormalized cluster assignment probabilities. The vector soft_z[n] can be converted back to a normalized simplex using the softmax function (see Section 31.6), either externally externally or within the model's generated quantities block.

Generalizing Soft K-Means

The multivariate normal distribution with unit covariance matrix produces a log probability density proportional to Euclidean distance (i.e., L_2 distance). Other distributions relate to other geometries. For instance, replacing the normal distribution with the double exponential (Laplace) distribution produces a clustering model based on L_1 distance (i.e., Manhattan or taxicab distance).

Within the multivariate normal version of K-means, replacing the unit covariance matrix with a shared covariance matrix amounts to working with distances defined in a space transformed by the inverse covariance matrix.

Although there is no global spatial analog, it is common to see soft K-means specified with a per-cluster covariance matrix. In this situation, a hierarchical prior may be used for the covariance matrices.

15.2. The Difficulty of Bayesian Inference for Clustering

Two problems make it pretty much impossible to perform full Bayesian inference for clustering models, the lack of parameter identifiability and the extreme multimodality of the posteriors.

Non-Identifiability

Cluster assignments are not identified — permuting the cluster mean vectors mu leads to a model with identical likelihoods. For instance, permuting the first two indexes in mu and the first two indexes in each soft_z[n] leads to an identical likelihood (and prior).

The lack of identifiability means that the the cluster parameters cannot be compared across multiple Markov chains. In fact, the only parameter in soft K-means is not identified, leading to problems in monitoring convergence. Clusters can even fail to be identified within a single chain, with indices swapping if the chain is long enough or the data is not cleanly separated.

Multimodality

The other problem with clustering models is that their posteriors are highly multimodal. One form of multimodality is the non-identifiability leading to index swapping. But even without the index problems the posteriors are highly mulitmodal.

Bayesian inference fails in cases of high multimodality because there is no way to visit all of the modes in the posterior in appropriate proportions and thus no way to evaluate integrals involved in posterior predictive inference.

In light of these two problems, the advice often given in fitting clustering models is to try many different initializations and select the sample with the highest overall probability. It is also popular to use optimization-based point estimators such as expectation maximization or variational Bayes, which can be much more efficient than sampling-based approaches.

15.3. Naive Bayes Classification and Clustering

Multinomial mixture models are referred to as "naive Bayes" because they are often applied to classification problems where the multinomial independence assumptions are clearly false.

Naive Bayes classification and clustering can be applied to any data with multinomial structure. A typical example of this is natural language text classification and clustering, which is used an example in what follows.

The observed data consists of a sequence of M documents made up of bags of words drawn from a vocabulary of V distinct words. A document m has N_m words, which are indexed as $w_{m,1}, \ldots, w_{m,N[m]} \in 1{:}V$. Despite the ordered indexing of words in a document, this order is not part of the model, which is clearly defective for natural human language data. A number of topics (or categories) K is fixed.

The multinomial mixture model generates a single category $z_m \in 1{:}K$ for each document $m \in 1{:}M$ according to a categorical distribution,

$$z_m \sim \textsf{Categorical}(\theta).$$

The K-simplex parameter θ represents the prevalence of each category in the data.

Next, the words in each document are generated conditionally independently of each other and the words in other documents based on the category of the document, with word n of document m being generated as

$$w_{m,n} \sim \textsf{Categorical}(\phi_{z[m]}).$$

The parameter $\phi_{z[m]}$ is a V-simplex representing the probability of each word in the vocabulary in documents of category z_m.

The parameters θ and π are typically given symmetric Dirichlet priors. The prevalence θ is sometimes fixed to produce equal probabilities for each category $k \in 1 : K$.

Representing Ragged Arrays in Stan

The specification for naive Bayes in the previous sections have used a ragged array notation for the words w. Because Stan does not support ragged arrays, the models are coded using an alternative strategy that provides an index for each word in a global list of words. The data is organized as follows, with the word arrays layed out in a column and each assigned to its document in a second column.

n	w[n]	doc[n]
1	$w_{1,1}$	1
2	$w_{1,2}$	1
$\vdots$	$\vdots$	$\vdots$
N_1	$w_{1,N[1]}$	1
$N_1 + 1$	$w_{2,1}$	2
$N_1 + 2$	$w_{2,2}$	2
$\vdots$	$\vdots$	$\vdots$
$N_1 + N_2$	$w_{2,N[2]}$	2
$N_1 + N_2 + 1$	$w_{3,1}$	3
$\vdots$	$\vdots$	$\vdots$
$\mathsf{N} = \sum_{m=1}^{M} N_m$	$w_{M,N[M]}$	M

The relevant variables for the program are N, the total number of words in all the documents, the word array w, and the document identity array doc.

Estimation with Category-Labeled Training Data

The naive Bayes models along with R programs to simulate data for them and a sample data set are available in the distribution in the directory src/models/misc/ clustering/naive-bayes.

A naive Bayes model for estimating the simplex parameters given training data with documents of known categories can be coded in Stan as follows

```
data {
  // training data
  int<lower=1> K;              // num topics
  int<lower=1> V;              // num words
  int<lower=0> M;              // num docs
  int<lower=0> N;              // total word instances
  int<lower=1,upper=K> z[M];   // topic for doc m
  int<lower=1,upper=V> w[N];   // word n
  int<lower=1,upper=M> doc[N]; // doc ID for word n
  // hyperparameters
  vector<lower=0>[K] alpha;    // topic prior
  vector<lower=0>[V] beta;     // word prior
}
parameters {
  simplex[K] theta;   // topic prevalence
  simplex[V] phi[K];  // word dist for topic k
}
model {
  theta ~ dirichlet(alpha);
  for (k in 1:K)
    phi[k] ~ dirichlet(beta);
  for (m in 1:M)
    z[m] ~ categorical(theta);
  for (n in 1:N)
    w[n] ~ categorical(phi[z[doc[n]]]);
}
```

Note that the topic identifiers z_m are declared as data and the latent category assignments are included as part of the likelihood function.

Estimation without Category-Labeled Training Data

Naive Bayes models can be used in an unsupervised fashion to cluster multinomial-structured data into a fixed number K of categories. The data declaration includes the same variables as the model in the previous section excluding the topic labels z. Because z is discrete, it needs to be summed out of the model calculation. This is done for naive Bayes as for other mixture models. The parameters are the same up to the priors, but the likelihood is now computed as the marginal document probability

$$
\begin{aligned}
&\log p(w_{m,1}, \ldots, w_{m,N_m} | \theta, \phi) \\
&= \log \sum_{k=1}^{K} \left(\text{Categorical}(k|\theta) \times \prod_{n=1}^{N_m} \text{Categorical}(w_{m,n}|\phi_k) \right) \\
&= \log \sum_{k=1}^{K} \exp \left(\log \text{Categorical}(k|\theta) + \sum_{n=1}^{N_m} \log \text{Categorical}(w_{m,n}|\phi_k) \right).
\end{aligned}
$$

The last step shows how the `log_sum_exp` function can be used to stabilize the numerical calculation and return a result on the log scale.

```
model {
  real gamma[M,K];
  theta ~ dirichlet(alpha);
  for (k in 1:K)
    phi[k] ~ dirichlet(beta);
  for (m in 1:M)
    for (k in 1:K)
      gamma[m,k] <- categorical_log(k,theta);
  for (n in 1:N)
    for (k in 1:K)
      gamma[doc[n],k] <- gamma[doc[n],k]
                         + categorical_log(w[n],phi[k]);
  for (m in 1:M)
    increment_log_prob(log_sum_exp(gamma[m]));
}
```

The local variable `gamma[m,k]` represents the value

$$
\gamma_{m,k} = \log \text{Categorical}(k|\theta) + \sum_{n=1}^{N_m} \log \text{Categorical}(w_{m,n}|\phi_k).
$$

Given γ, the posterior probability that document m is assigned category k is

$$
\Pr[z_m = k | w, \alpha, \beta] = \exp \left(\gamma_{m,k} - \log \sum_{k=1}^{K} \exp(\gamma_{m,k}) \right).
$$

If the variable `gamma` were declared and defined in the transformed parameter block, its sampled values would be saved by Stan. The normalized posterior probabilities could also be defined as generated quantities.

Full Bayesian Inference for Naive Bayes

Full Bayesian posterior predictive inference for the naive Bayes model can be implemented in Stan by combining the models for labeled and unlabeled data. The estimands include both the model parameters and the posterior distribution over categories for the unlabeled data. The model is essentially a missing data model assuming the unknown category labels are missing completely at random; see (Gelman et al., 2003; Gelman and Hill, 2007) for more information on missing data imputation. The model is also an instance of semisupervised learning because the unlabeled data contributes to the parameter estimations.

To specify a Stan model for performing full Bayesian inference, the model for labeled data is combined with the model for unlabeled data. A second document collection is declared as data, but without the category labels, leading to new variables `M2 N2`, `w2`, `doc2`. The number of categories and number of words, as well as the hyperparameters are shared and only declared once. Similarly, there is only one set of parameters. Then the model contains a single set of statements for the prior, a set of statements for the labeled data, and a set of statements for the unlabeled data.

Prediction without Model Updates

An alternative to full Bayesian inference involves estimating a model using labeled data, then applying it to unlabeled data without updating the parameter estimates based on the unlabeled data. This behavior can be implemented by moving the definition of `gamma` for the unlabeled documents to the generated quantities block. Because the variables no longer contribute to the log probability, they no longer jointly contribute to the estimation of the model parameters.

15.4.   Latent Dirichlet Allocation

Latent Dirichlet allocation (LDA) is a mixed-membership multinomial clustering model (Blei et al., 2003) that generalized naive Bayes. Using the topic and document terminology common in discussions of LDA, each document is modeled as having a mixture of topics, with each word drawn from a topic based on the mixing proportions.

The LDA Model

The basic model assumes each document is generated independently based on fixed hyperparameters. For document m, the first step is to draw a topic distribution simplex θ_m over the K topics,

$$\theta_m \sim \text{Dirichlet}(\alpha).$$

The prior hyperparameter α is fixed to a K-vector of positive values. Each word in the document is generated independently conditional on the distribution θ_m. First, a topic $z_{m,n} \in 1{:}K$ is drawn for the word based on the document-specific topic-distribution,

$$z_{m,n} \sim \mathsf{Categorical}(\theta_m).$$

Finally, the word $w_{m,n}$ is drawn according to the word distribution for topic $z_{m,n}$,

$$w_{m,n} \sim \mathsf{Categorical}(\phi_{z[m,n]}).$$

The distributions ϕ_k over words for topic k are also given a Dirichlet prior,

$$\phi_k \sim \mathsf{Dirichlet}(\beta)$$

where β is a fixed V-vector of positive values.

Summing out the Discrete Parameters

Although Stan does not (yet) support discrete sampling, it is possible to calculate the marginal distribution over the continuous parameters by summing out the discrete parameters as in other mixture models. The marginal posterior of the topic and word variables is

$$
\begin{aligned}
p(\theta, \phi | w, \alpha, \beta) \quad &\propto \quad p(\theta | \alpha) \times p(\phi | \beta) \times p(w | \theta, \phi) \\
&= \quad \prod_{m=1}^{M} p(\theta_m | \alpha) \times \prod_{k=1}^{K} p(\phi_k | \beta) \times \prod_{m=1}^{M} \prod_{n=1}^{M[n]} p(w_{m,n} | \theta_m, \phi).
\end{aligned}
$$

The inner word-probability term is defined by summing out the topic assignments,

$$
\begin{aligned}
p(w_{m,n} | \theta_m, \phi) \quad &= \quad \sum_{z=1}^{K} p(z, w_{m,n} | \theta_m, \phi). \\
&= \quad \sum_{z=1}^{K} p(z | \theta_m) \times p(w_{m,n} | \phi_z).
\end{aligned}
$$

Plugging the distributions in and converting to the log scale provides a formula that can be implemented directly in Stan,

$$
\begin{aligned}
&\log p(\theta, \phi | w, \alpha, \beta) \\
&= \sum_{m=1}^{M} \log \mathsf{Dirichlet}(\theta_m | \alpha) \; + \; \sum_{k=1}^{K} \log \mathsf{Dirichlet}(\phi_k | \beta) \\
&\quad + \sum_{m=1}^{M} \sum_{n=1}^{N[m]} \log \left(\sum_{z=1}^{K} \mathsf{Categorical}(z | \theta_m) \times \mathsf{Categorical}(w_{m,n} | \phi_z) \right)
\end{aligned}
$$

Implementation of LDA

Applying the marginal derived in the last section to the data structure described in this section leads to the following Stan program for LDA.

```
data {
  int<lower=2> K;               // num topics
  int<lower=2> V;               // num words
  int<lower=1> M;               // num docs
  int<lower=1> N;               // total word instances
  int<lower=1,upper=V> w[N];    // word n
  int<lower=1,upper=M> doc[N];  // doc ID for word n
  vector<lower=0>[K] alpha;     // topic prior
  vector<lower=0>[V] beta;      // word prior
}
parameters {
  simplex[K] theta[M];    // topic dist for doc m
  simplex[V] phi[K];      // word dist for topic k
}
model {
  for (m in 1:M)
    theta[m] ~ dirichlet(alpha);  // prior
  for (k in 1:K)
    phi[k] ~ dirichlet(beta);     // prior
  for (n in 1:N) {
    real gamma[K];
    for (k in 1:K)
      gamma[k] <- log(theta[doc[n],k]) + log(phi[k,w[n]]);
    increment_log_prob(log_sum_exp(gamma));  // likelihood
  }
}
```

As in the other mixture models, the log-sum-of-exponents function is used to stabilize the numerical arithmetic.

Correlated Topic Model

To account for correlations in the distribution of topics for documents, (Blei and Lafferty, 2007) introduced a variant of LDA in which the Dirichlet prior on the per-document topic distribution is replaced with a multivariate logistic normal distribution.

The authors treat the prior as a fixed hyperparameter. They use an L_1-regularized estimate of covariance, which is equivalent to the maximum a posteriori estimate given a double-exponential prior. Stan does not (yet) support maximum a posteriori

estimation, so the mean and covariance of the multivariate logistic normal must be specified as data.

Fixed Hyperparameter Correlated Topic Model

The Stan model in the previous section can be modified to implement the correlated topic model by replacing the Dirichlet topic prior `alpha` in the data declaration with the mean and covariance of the multivariate logistic normal prior.

```
data {
  ... data as before without alpha ...
  vector[K] mu;          // topic mean
  cov_matrix[K] Sigma;   // topic covariance
}
```

Rather than drawing the simplex parameter `theta` from a Dirichlet, a parameter `eta` is drawn from a multivariate normal distribution and then transformed using softmax into a simplex.

```
parameters {
  simplex[V] phi[K];  // word dist for topic k
  vector[K] eta[M];   // topic dist for doc m
}
transformed parameters {
  simplex[K] theta[M];
  for (m in 1:M)
    theta[m] <- softmax(eta[m]);
}
model {
  for (m in 1:M)
    eta[m] ~ multi_normal(mu,Sigma);
  ... model as before w/o prior for theta ...
}
```

Full Bayes Correlated Topic Model

By adding a prior for the mean and covariance, Stan supports full Bayesian inference for the correlated topic model. This requires moving the declarations of topic mean `mu` and covariance `Sigma` from the data block to the parameters block and providing them with priors in the model. A relatively efficient and interpretable prior for the covariance matrix `Sigma` may be encoded as follows.

```
... data block as before, but without alpha ...
parameters {
```

```
      vector[K] mu;              // topic mean
      corr_matrix[K] Omega;      // correlation matrix
      vector<lower=0>[K] sigma;  // scales
      vector[K] eta[M];          // logit topic dist for doc m
      simplex[V] phi[K];         // word dist for topic k
    }
    transformed parameters {
      ... eta as above ...
      cov_matrix[K] Sigma;       // covariance matrix
      for (m in 1:K)
        Sigma[m,m] <- sigma[m] * sigma[m] * Omega[m,m];
      for (m in 1:(K-1)) {
        for (n in (m+1):K) {
          Sigma[m,n] <- sigma[m] * sigma[n] * Omega[m,n];
          Sigma[n,m] <- Sigma[m,n];
        }
      }
    }
    model {
      mu ~ normal(0,5);        // vectorized, diffuse
      Omega ~ lkj_corr(2.0);   // regularize to unit correlation
      sigma ~ cauchy(0,5);     // half-Cauchy due to constraint
      ... words sampled as above ...
    }
```

The LkjCorr distribution with shape $\alpha > 0$ has support on correlation matrices (i.e., symmetric positive definite with unit diagonal). Its density is defined by

$$\mathsf{LkjCorr}(\Omega|\alpha) \propto \det(\Omega)^{\alpha-1}$$

With a scale of $\alpha = 2$, the weakly informative prior favors a unit correlation matrix. Thus the compound effect of this prior on the covariance matrix Σ for the multivariate logistic normal is a slight concentration around diagonal covariance matrices with scales determined by the prior on sigma.

16.   Gaussian Processes

Gaussian process are continuous stochastic processes and thus may be interpreted as providing a probability distribution over functions. A probability distribution over continuous functions may be viewed, roughly, as an uncountably infinite collection of random variables, one for each valid input. The generality of the supported functions makes Gaussian priors popular choices for priors in general multivariate (non-linear) regression problems.

The defining feature of a Gaussian process is that the distribution of the function's value at a finite number of input points is a multivariate normal distribution. This makes it tractable to both fit models from finite amounts of observed data and make predictions for finitely many new data points.

Unlike a simple multivariate normal distribution, which is parameterized by a mean vector and covariance matrix, a Gaussian process is parameterized by a mean function and covariance function. The mean and covariance functions apply to vectors of inputs and return a mean vector and covariance matrix which provide the mean and covariance of the outputs corresponding to those input points in the functions drawn from the process.

Gaussian processes can be encoded in Stan by implementing their mean and covariance functions and plugging the result into the Gaussian form of their sampling distribution. This form of model is easy to understand and may be used for simulation, model fitting, or posterior predictive inference. More efficient Stan implementation for the basic (non-logistic) regression applies a Cholesky-factor reparameterization of the Gaussian and computes the posterior predictive distribution analytically.

After defining Gaussian processes, this chapter covers the basic implementations for simulation, hyperparameter estimation, and posterior predictive inference for univariate regressions, multivariate regressions, and multivariate logistic regressions. Gaussian processes are very general, and by necessity this chapter only touches on some basic models. For more information, see (Rasmussen and Williams, 2006).

16.1.   Gaussian Process Regression

The data for a multivariate Gaussian process regression consists of a series of N inputs $x_1, \ldots, x_N \in \mathbb{R}^D$ paired with outputs $y_1, \ldots, y_N \in \mathbb{R}$. The defining feature of Gaussian processes is that the probability of a finite number of outputs y conditioned on their inputs x is Gaussian,

$$y \sim \mathsf{Normal}(m(x), k(x)),$$

where $m(x)$ is an N-vector and and $k(x)$ is an $N \times N$ covariance matrix. The mean function $m : \mathbb{R}^{N \times D} \rightarrow \mathbb{R}^N$ can be anything, but the covariance function $k : \mathbb{R}^{N \times D} \rightarrow \mathbb{R}^{N \times N}$ must produce a positive-definite matrix for any input x.[1]

A popular covariance function, which will be used in the implementations later in this chapter, is a generalized, squared exponential function,

$$k(x)_{i,j} = \eta^2 \exp \left(-\rho^2 \sum_{d=1}^{D} (x_{i,d} - x_{j,d})^2 \right) + \delta_{i,j} \sigma^2,$$

where η, ρ, and σ are hyperparameters defining the covariance function and where $\delta_{i,j}$ is the Kronecker delta function with value 1 if $i = j$ and value 0 otherwise; note that this test is between the indexes i and j, not between values x_i and x_j. The addition of σ^2 on the diagonal is import to ensure the positive definiteness of the resulting matrix in the case of two identical inputs $x_i = x_j$. In statistical terms, σ is the scale of the noise term in the regression.

The only term in the squared exponential covariance function involving the inputs x_i and x_j is their vector difference, $x_i - x_j$. This produces a process with stationary covariance in the sense that if an input vector x is translated by a vector ϵ to $x + \epsilon$, the covariance at any pair of outputs is unchanged, because $k(x) = k(x + \epsilon)$.

The summation involved is just the squared Euclidean distance between x_i and x_j (i.e., the L_2 norm of their difference, $x_i - x_j$). This results in support for smooth functions in the process. The amount of variation in the function is controlled by the free hyperparameters η, ρ, and σ.

Changing the notion of distance from Euclidean to taxicab distance (i.e., an L_1 norm) changes the support to functions which are continuous but not smooth.

16.2.   Simulating from a Gaussian Process

It is simplest to start with a Stan model that does nothing more than simulate draws of functions f from a Gaussian process. In practical terms, the model will draw values $y_n = f(x_n)$ for finitely many input points x_n.

The Stan model defines the mean and covariance functions in a transformed data block and then samples outputs y in the model using a multivariate normal distribution. To make the model concrete, the squared exponential covariance function described in the previous section will be used with hyperparameters set to $\eta^2 = 1$, $\rho^2 = 1$, and $\sigma^2 = 0.1$, and the mean function m is defined to always return the zero vector, $m(x) = \mathbf{0}$. The following model is included in the Stan distribution in file `src/models/misc/gaussian-process/gp-sim.stan`.

[1]Gaussian processes can be extended to covariance functions producing positive semi-definite matrices, but Stan does not support inference in the resulting models because the resulting distribution does not have unconstrained support.

```
data {
  int<lower=1> N;
  real x[N];
}
transformed data {
  vector[N] mu;
  cov_matrix[N] Sigma;
  for (i in 1:N)
    mu[i] <- 0;
  for (i in 1:N)
    for (j in 1:N)
      Sigma[i,j] <- exp(-pow(x[i] - x[j],2))
                    + if_else(i==j, 0.1, 0.0);
}
parameters {
  vector[N] y;
}
model {
  y ~ multi_normal(mu,Sigma);
}
```

The input data is just the vector of inputs x and its size N. Such a model can be used with values of x evenly spaced over some interval in order to plot sample draws of functions from a Gaussian process. The covariance matrix Sigma is not being computed efficiently here; see Section Section 16.3 for a better approach.

Multivariate Inputs

Only the covariance function's distance computation needs to change in moving from a univariate model to a multivariate model. A multivariate sampling model is available in the source distribution at src/models/misc/gaussian-process/ gp-multi-sim.stan. The only lines that change from the univariate model above are as follows.

```
data {
  int<lower=1> D;
  int<lower=1> N;
  vector[D] x[N];
}
transformed data {
...
      Sigma[i,j] <- exp(-dot_self(x[i] - x[j]))
                    + if_else(i==j, 0.1, 0.0);
...
```

The data is now declared as an array of vectors instead of an array of scalars; the dimensionality D is also declared. The squared Euclidean distance calculation is done using the `dot_self` function, which returns the dot product of its argument with itself, here `x[i] - x[j]`.

In the remainder of the chapter, univariate models will be used for simplicity, but any of them could be changed to multivariate in the same way as the simple sampling model. The only extra computational overhead from a multivariate model is in the distance calculation, which is only done once when the transformed data block is run after the data is read.

Cholesky Factored and Transformed Implementation

A much more efficient implementation of the simulation model can be coded in Stan by relocating, rescaling and rotating an isotropic unit normal variate. Suppose z is an an isotropic unit normal variate

$$z \sim \mathsf{Normal}(\mathbf{0}, \mathbf{1}),$$

where $\mathbf{0}$ is an N-vector of 0 values and $\mathbf{1}$ is the $N \times N$ unit matrix. Let L be the the Cholesky decomposition of $k(x)$, i.e., the lower-triangular matrix L such that $LL^\top = k(x)$. Then the transformed variable $\mu + Lz$ has the intended target distribution,

$$\mu + Lz \sim \mathsf{Normal}(\mu, k(x)).$$

This transform can be applied directly to Gaussian process simulation, as shown in the model `src/models/misc/gaussian-process/gp-sim-cholesky.stan` in the distribution. This model has the same data declarations for N and x, and the same transformed data definitions of mu and Sigma as the previous model, with the addition of a transformed data variable for the Cholesky decomposition. The parameters change to the raw parameters sampled from an isotropic unit normal, and the actual samples are defined as generated quantities.

```
...
transformed data {
  matrix[N,N] L;
...
  L <- cholesky_decompose(Sigma);
}
parameters {
  vector[N] z;
}
model {
  z ~ normal(0,1);
```

137

```
  }
  generated quantities {
    vector[N] y;
    y <- mu + L * z;
  }
```

The Cholesky decomposition is only computed once, after the data is loaded and the covariance matrix `Sigma` computed. The isotropic normal distribution for `z` is specified as a vectorized univariate distribution for efficiency; this specifies that each `z[n]` has an independent unit normal distribution. The sampled vector `y` is then defined as a generated quantity using a direct encoding of the transform described above.

16.3. Fitting a Gaussian Process

The hyperparameters controlling the covariance function of a Gaussian process can be fit by assigning them priors, then computing the posterior distribution of the hyperparameters given observed data. Because the hyperparameters are required to be positive and expected to have reasonably small values, broad half-Cauchy distributions act as quite vague priors which could just as well be uniform over a constrained range of values. The priors on the parameters should be defined based on prior knowledge of the scale of the output values (η), the scale of the output noise (σ), and the scale at which distances are measured among inputs ($1/\rho$).

A Stan model to fit the hyperparameters of the general squared exponential covariance function is provided in the distribution in `src/models/misc/gaussian-process/gp-fit.stan`. The Stan code is very similar to the simulation models in terms of the computations, but the blocks in which variables are declared and statements are executed has changed to accommodate the hyperparameter estimation problem.

```
  data {
    int<lower=1> N;
    vector[N] x;
    vector[N] y;
  }
  transformed data {
    vector[N] mu;
    for (i in 1:N) mu[i] <- 0;
  }
  parameters {
    real<lower=0> eta_sq;
    real<lower=0> rho_sq;
```

```
      real<lower=0> sigma_sq;
    }
    model {
      matrix[N,N] Sigma;
      // off-diagonal elements
      for (i in 1:(N-1)) {
        for (j in i:N) {
          Sigma[i,j] <- eta_sq * exp(-rho_sq * pow(x[i] - x[j],2));
          Sigma[j,i] <- Sigma[i,j];
        }
      }
      // diagonal elements
      for (k in 1:N)
        Sigma[k,k] <- eta_sq + sigma_sq;  // + jitter

      eta_sq ~ cauchy(0,5);
      rho_sq ~ cauchy(0,5);
      sigma_sq ~ cauchy(0,5);

      y ~ multi_normal(mu,Sigma);
    }
```

The data block now declares a vector y of observed values y[n] for inputs x[n].
The transformed data block now only defines the mean vector to be zero. The three
hyperparameters are defined as parameters constrained to be non-negative. The com-
putation of the covariance matrix Sigma is now in the model block because it involves
unknown parameters and thus can't simply be precomputed as transformed data. The
rest of the model consists of the priors for the hyperparameters and the multivariate
normal likelihood, only now the value y is known and the covariance matrix Sigma is
an unknown dependent on the hyperparameters.

Hamiltonian Monte Carlo sampling is quite fast and effective for hyperparameter
inference in this model (Neal, 1997), and the Stan implementation will fit hyperpa-
rameters in models with hundreds of data points in seconds.

Automatic Relevance Determination

For multivariate inputs $x \in \mathbb{R}^D$, the squared exponential covariance function can be
further generalized by fitting a precision parameter ρ_d^2 for each dimension d,

$$k(x)_{i,j} = \eta^2 \exp\left(-\sum_{d=1}^{D} \rho_d^2 (x_{i,d} - x_{j,d})^2\right) + \delta_{i,j}\sigma^2.$$

The estimation of ρ was termed "automatic relevance determination" in (Neal, 1996),
because the larger ρ_d is, the more dimension d is weighted in the distance calculation.

The implementation of automatic relevance determination in Stan is straightforward. A model like the one to fit the basic hyperparameters can be generalized by declaring rho to be a vector of size D and defining the covariance function as in this subsection.

The collection of ρ_d parameters can also be modeled hierarchically.

16.4. Predictive Inference with a Gaussian Process

Suppose for a given sequence of inputs x that the corresponding outputs y are observed. Given a new sequence of inputs $\tilde{x}$, the posterior predictive distribution of their labels is computed by sampling outputs $\tilde{y}$ according to

$$p(\tilde{y}|\tilde{x},x,y) \ = \ \frac{p(\tilde{y},y|\tilde{x},x)}{p(y|x)} \ \propto \ p(\tilde{y},y|\tilde{x},x).$$

A direct implementation in Stan defines a model in terms of the the joint distribution of the observed y and unobserved $\tilde{y}$. Although Stan does not support mixed vectors of parameters and data directly, such a vector may be synthesized as a local variable in the model block. The following model, which takes this approach, is available in the distribution as src/models/misc/gaussian-process/gp-predict.stan.

```
data {
  int<lower=1> N1;
  vector[N1] x1;
  vector[N1] y1;
  int<lower=1> N2;
  vector[N2] x2;
}
transformed data {
  int<lower=1> N;
  vector[N1+N2] x;
  vector[N1+N2] mu;
  cov_matrix[N1+N2] Sigma;
  N <- N1 + N2;
  for (n in 1:N1) x[n] <- x1[n];
  for (n in 1:N2) x[N1 + n] <- x2[n];
  for (i in 1:N) mu[i] <- 0;
  for (i in 1:N)
    for (j in 1:N)
      Sigma[i,j] <- exp(-pow(x[i] - x[j],2))
                    + if_else(i==j, 0.1, 0.0);
}
parameters {
```

```
    vector[N2] y2;
}
model {
  vector[N] y;
  for (n in 1:N1) y[n] <- y1[n];
  for (n in 1:N2) y[N1 + n] <- y2[n];

  y ~ multi_normal(mu,Sigma);
}
```

The input vectors x1 and x2 are declared as data, as is the observed output vector y1. The unknown output vector y2, which corresponds to input vector x2, is declared as a parameter and will be sampled when the model is executed.

A transformed data block is used to combine the input vectors x1 and x2 into a single vector x. The covariance function is then applied to this combined input vector to produce the covariance matrix Sigma. The mean vector mu is also declared and set to zero.

The model block declares and define a local variable for the combined output vector y, which consists of the concatenation of the known outputs y1 and unknown outputs y2. Thus the combined output vector y is aligned with the combined input vector x. All that is left is to define the multivariate normal sampling statement for y.

Cholesky Factorization Speedup

This model could be sped up fairly substantially by computing the Cholesky factor of Sigma in the transformed data block

```
transformed data {
  matrix[N1+N2,N1+N2] L;
...
  L = cholesky_decompose(Sigma);
...
```

and then replacing multi_normal with the more efficient multi_normal_cholesky in the model block.

```
...
model {
...
  y ~ multi_normal_cholesky(mu,L);
}
```

At this point, Sigma could be declared as a local variable in the data block so that its memory may be recovered after the data is loaded.

Analytical Form of Joint Predictive Inference

Bayesian predictive inference for Gaussian processes can be sped up by deriving the posterior analytically, then directly sampling from it. This works for standard Gaussian processes, but not generalizations such as logistic Gaussian process regression.

Jumping straight to the result,

$$p(\tilde{y}|\tilde{x}, y, x) = \mathsf{Normal}(K^\top \Sigma^{-1} y, \ \Omega - K^\top \Sigma^{-1} K),$$

where $\Sigma = k(x)$ is the result of applying the covariance function to the inputs x with observed outputs y, $\Omega = k(\tilde{x})$ is the result of applying the covariance function to the inputs $\tilde{x}$ for which predictions are to be inferred, and K is the matrix of covariances between inputs x and $\tilde{x}$, which in the case of the generalized squared exponential covariance function would be

$$K_{i,j} = \eta^2 \exp(-\rho^2 \sum_{d=1}^{D} (x_{i,d} - \tilde{x}_{j,d})^2).$$

There is no noise term including σ^2 because the indexes of elements in x and $\tilde{x}$ are never the same.

Because a Stan model is only required to be proportional to the posterior, the posterior may be coded directly. An example that uses the analytic form of the posterior and provides sampling of the resulting multivariate normal through the Cholesky decomposition is provided in `src/models/misc/gaussian-process/gp-predict-analytic.stan`. The data declaration is the same as for the standard example. The calculation of the predictive mean `mu` and covariance Cholesky factor `L` is done in the transformed data block.

```
transformed data {
  vector[N2] mu;
  matrix[N2,N2] L;
  {
    matrix[N1,N1] Sigma;
    matrix[N2,N2] Omega;
    matrix[N1,N2] K;

    matrix[N2,N1] K_transpose_div_Sigma;
    matrix[N2,N2] Tau;

    for (i in 1:N1)
      for (j in 1:N1)
        Sigma[i,j] <- exp(-pow(x1[i] - x1[j],2))
          + if_else(i==j, 0.1, 0.0);
    for (i in 1:N2)
```

```
      for (j in 1:N2)
        Omega[i,j] <- exp(-pow(x2[i] - x2[j],2))
          + if_else(i==j, 0.1, 0.0);
    for (i in 1:N1)
      for (j in 1:N2)
        K[i,j] <- exp(-pow(x1[i] - x2[j],2));

    K_transpose_div_Sigma <- K' / Sigma;
    mu <- K_transpose_div_Sigma * y1;
    Tau <- Omega - K_transpose_div_Sigma * K;
    for (i in 1:(N2-1))
      for (j in (i+1):N2)
        Tau[i,j] <- Tau[j,i];

    L <- cholesky_decompose(Tau);
  }
}
```

This block implements the definitions of Σ, Ω, and K directly. The posterior mean vector $K^\top \Sigma^{-1} y$ is computed as mu. The covariance has a Cholesky factor L such that $LL^\top = \Omega - K^\top \Sigma^{-1} K$. Given these two ingredients, sampling the predictive quantity $\tilde{y}$ is carried out by translating, scaling and rotating an isotropic normal sample using the posterior mean and the Cholesky factorization of the posterior covariance.

Joint Hyperparameter Fitting and Predictive Inference

Hyperparameter fitting may be carried out jointly with predictive inference in a single model. This allows full Bayesian inference to account for the affect of the uncertainty in the hyperparameter estimates on the predictive inferences.

To encode a joint hyperparameter fit and predictive inference model in Stan, declare the hyperparameters as additional parameters, give them a prior in the model, move the definition of Sigma to a local variable in the model defined using the hyperparameters.

16.5. Classification with Gaussian Processes

Gaussian processes can be generalized the same way as standard linear models by introducing a link function. This allows them to be used as discrete data models, and in particular to perform classification using posterior predictive inference. This section focuses on binary classification problems implemented with logistic Gaussian process regression.

Logistic Gaussian Process Regression

For binary classification problems, the observed outputs $z_n \in \{0, 1\}$ are binary. These outputs are modeled using a Gaussian process with (unobserved) outputs y_n through the logistic link,

$$z_n \sim \text{Bernoulli}(\text{logit}^{-1}(y_n)),$$

or in other words,

$$\Pr[z_n = 1] = \text{logit}^{-1}(y_n).$$

Simulation

Simulation from a Gaussian process logistic regression is straightforward; just simulate from a Gaussian process and then simulate the z_n from the y_n using the sampling distribution above. This cannot be done directly in Stan because Stan does not (yet) support discrete parameters or forward discrete sampling.

Hyperparameter Estimation and Predictive Inference

For hyperparameter estimation and predictive inference applications, the y_n are typically latent parameters (i.e., not observed). Unfortunately, they cannot be easily marginalized out analytically, so they must be estimated from the data through the observed categorical outputs z_n. Predictive inference will proceed not by sampling z_n values, but directly through their probabilities, given by $\text{logit}^{-1}(y_n)$.

Stan Implementations

Hyperparameter estimation and predictive inference are easily accomplished in Stan by declaring the vector y as a parameter, adding the sampling statements for observed z, and then proceeding as for the previous regression models.

The following full model for prediction using logistic Gaussian process regression is available in the distribution at `src/models/misc/gaussian-process/gp-logit-predict.stan`.

```
data {
  int<lower=1> N1;
  vector[N1] x1;
  int<lower=0,upper=1> z1[N1];
  int<lower=1> N2;
  vector[N2] x2;
}
transformed data {
 ... define mu as zero, compute Sigma from x1, x2 ...
```

```
    }
    parameters {
      vector[N1] y1;
      vector[N2] y2;
    }
    model {
      vector[N] y;
      for (n in 1:N1) y[n] <- y1[n];
      for (n in 1:N2) y[N1 + n] <- y2[n];

      y ~ multi_normal(mu,Sigma);
      for (n in 1:N1)
        z1[n] ~ bernoulli_logit(y1[n]);
    }
```

The transformed data block in which `mu` and `Sigma` are defined is not shown because it is identical to the model for prediction in the previous section. Now the observed outcomes `z1`, declared as data, are binary. The variable `y1` is still drawn from the Gaussian process with values `y1[n]` being the values of the function for input `x1[n]`, only now `y1[n]` is interpreted as the logit-scaled probability that `z1[n]` is 1. The variable `y2` plays the same role for probabilistic predictions for inputs `x2` and is also declared as a parameter.

In the model, the full vector `y` is defined as before by concatenating `y1` and `y2`, only this time both `y1` and `y2` are parameters. The full vector `y` is defined as being multivariate normal as before. Additionally, the `z1[n]` variables are given a Bernoulli distribution with logit-scaled parameters. Only the `z1[n]` values are observed and hence only they are sampled. There is no `z2[n]` vector because Stan does not support discrete sampling; instead, the predictions are in the form of the logit-scaled probabilities `y2`.

Samples form this model do not mix as well as for the standard model. This is largely because the `z1` values are quantized forms of `y1`, and thus provide less precise data for estimation.

The model could be sped up by applying a Cholesky decomposition to the covariance matrix `Sigma` and then replacing the `multi_normal` distribution with `multi_normal_cholesky`.

A pure logistic Gaussian process regression would not include a noise term in the definition of the covariance matrix. This can be implemented by simply removing the noise term(s) `sigma_sq` from the definition of `Sigma`. Probit regression can be coded by subsituting the probit link for the logit.[2]

---

[2]Although it is possible to implement probit regression by including the noise term `sigma_sq` and then quantizing `y1[n]` to produce `z1[n]`, this is not feasible in Stan because it requires a complex constraint on `y` to be enforced for multivariate normal distribution.

This simple prediction model could be extended in the same way as previous models by declaring the hyperparameters as parameters and defining the covariance matrix in the model block as a local variable.

17.   Reparameterization & Change of Variables

As with BUGS, Stan supports a direct encoding of reparameterizations. Stan also supports changes of variables by directly incrementing the log probability accumulator with the log Jacobian of the transform.

17.1.   Reparameterizations

Reparameterizations may be implemented straightforwardly. For example, the Beta distribution is parameterized by two positive count parameters $\alpha, \beta > 0$. The following example illustrates a hierarchical Stan model with a vector of parameters `theta` are drawn i.i.d. for a Beta distribution whose parameters are themselves drawn from a hyperprior distribution.

```
parameters {
  real<lower = 0> alpha;
  real<lower = 0> beta;
  ...
model {
  alpha ~ ...
  beta ~ ...
  for (n in 1:N)
    theta[n] ~ beta(alpha,beta);
  ...
```

It is often more natural to specify hyperpriors in terms of transformed parameters. In the case of the Beta, the obvious choice for reparameterization is in terms of a mean parameter

$$\phi = \alpha/(\alpha + \beta)$$

and total count parameter

$$\lambda = \alpha + \beta.$$

Following (Gelman et al., 2003, Chapter 5), the mean gets a uniform prior and the count parameter a Pareto prior with $p(\lambda) \propto \lambda^{-2.5}$.

```
parameters {
  real<lower=0,upper=1> phi;
  real<lower=0.1> lambda;
  ...
transformed parameters {
  real<lower=0> alpha;
  real<lower=0> beta;
```

```
      ...
      alpha <- lambda * phi;
      beta <- lambda * (1 - phi);
      ...
    model {
      phi ~ beta(1,1); // uniform on phi, could drop
      lambda ~ pareto(0.1,1.5);
      for (n in 1:N)
        theta[n] ~ beta(alpha,beta);
      ...
```

The new parameters, `phi` and `lambda`, are declared in the parameters block and the parameters for the Beta distribution, `alpha` and `beta`, are declared and defined in the transformed parameters block. And If their values are not of interest, they could instead be defined as local variables in the model as follows.

```
    model {
      real alpha;
      real beta;
      alpha <- lambda * phi;
      beta <- lambda * (1 - phi);
...
      for (n in 1:N)
        theta[n] ~ beta(alpha,beta);
...
    }
```

With vectorization, this could be expressed more compactly and efficiently as follows.

```
    model {
      theta ~ beta(lambda * phi, lambda * (1 - phi));
...
    }
```

If the variables `alpha` and `beta` are of interest, they can be defined in the transformed parameter block and then used in the model.

Jacobians not Necessary

Because the transformed parameters are being used, rather than given a distribution, there is no need to apply a Jacobian adjustment for the transform. For example, in the beta distribution example, `alpha` and `beta` have the correct posterior distribution.

17.2. Changes of Variables

Changes of variables are applied when the transformation of a parameter is characterized by a distribution. The standard textbook example is the lognormal distribution, which is the distribution of a variable $y > 0$ whose logarithm $\log y$ has a normal distribution. Note that the distribution is being assigned to $\log y$.

The change of variables requires an adjustment to the probability to account for the distortion caused by the transform. For this to work, univariate changes of variables must be monotonic and differentiable everywhere in their support.

For univariate changes of variables, the resulting probability must be scaled by the absolute derivative of the transform (see Section 49.1 for more precise definitions of univariate changes of variables).

In the case of log normals, if y's logarithm is normal with mean μ and deviation σ, then the distribution of y is given by

$$p(y) \; = \; \mathsf{Normal}(\log y \,|\, \mu, \sigma) \, \left| \frac{d}{dy} \log y \right| \; = \; \mathsf{Normal}(\log y \,|\, \mu, \sigma) \frac{1}{y}.$$

Stan works on the log scale to prevent underflow, where

$$\log p(y) = \log \mathsf{Normal}(\log y \,|\, \mu, \sigma) - \log y.$$

In Stan, the change of variables can be applied in the sampling statement. To adjust for the curvature, the log probability accumulator is incremented with the log absolute derivative of the transform. The lognormal distribution can thus be implemented directly in Stan as follows.[1]

```
parameters {
  real<lower=0> y;
  ...
model {
  log(y) ~ normal(mu,sigma);
  increment_log_prob(- log(y));
  ...
```

It is important, as always, to declare appropriate constraints on parameters; here y is constrained to be positive.

It would be slightly more efficient to define a local variable for the logarithm, as follows.

---

[1]This example is for illustrative purposes only; the recommended way to implement the lognormal distribution in Stan is with the built-in `lognormal` probability function (see Section 37.1).

```
model {
  real log_y;
  log_y <- log(y);
  log_y ~ normal(mu,sigma);
  increment_log_prob(- log_y);
  ...
```

If y were declared as data instead of as a parameter, then the adjustment can be ignored because the data will be constant and Stan only requires the log probability up to a constant.

Change of Variables vs. Transformations

This section illustrates the difference between a change of variables and a simple variable transformation. A transformation samples a parameter, then transforms it, whereas a change of variables transforms a parameter, then samples it. Only the latter requires a Jacobian adjustment.

Gamma and Inverse Gamma Distribution

Like the log normal, the inverse gamma distribution is a distribution of variables whose inverse has a gamma distribution. This section contrasts two approaches, first with a transform, then with a change of variables.

The transform based approach to sampling y_inv with an inverse gamma distribution can be coded as follows.

```
parameters {
  real<lower=0> y;
}
transformed parameters {
  real<lower=0> y_inv;
  y_inv <- 1 / y;
}
model {
  y ~ gamma(2,4);
}
```

The change-of-variables approach to sampling y_inv with an inverse gamma distribution can be coded as follows.

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

```
}
transformed parameters {
  real<lower=0> y;
  y <- 1 / y_inv;                            // change
  increment_log_prob( -2 * log(y_inv) );  // adjustment
}
model {
  y ~ gamma(2,4);
}
```

The Jacobian adjustment is the log of the absolute derivative of the transform, which in this case is

$$
\begin{aligned}
\log \left| \frac{d}{du} \left(\frac{1}{u} \right) \right| &= \log \frac{1}{u^{-2}} \\
&= -2 \log u.
\end{aligned}
$$

Multivariate Changes of Variables

In the case of a multivariate transform, the log of the Jacobian of the transform must be added to the log probability accumulator (see the subsection of Section 49.1 on multivarate changes of variables for more precise definitions of multivariate transforms and Jacobians). In Stan, this can be coded as follows in the general case where the Jacobian is not a full matrix.

```
parameters {
  vector[K] u;      // multivariate parameter
   ...
transformed parameters {
  vector[K] v;     // transformed parameter
  matrix[K,K] J;   // Jacobian matrix of transform
  ... compute v as a function of u ...
  ... compute J[m,n] = d.v[m] / d.u[n] ...
  increment_log_prob(log(fabs(determinant(J))));
  ...
model {
  v ~ ...;
  ...
```

Of course, if the Jacobian is known analytically, it will be more efficient to apply it directly than to call the determinant function, which is neither efficient nor particularly stable numerically.

In many cases, the Jacobian matrix will be triangular, so that only the diagonal elements will be required for the determinant calculation. Triangular Jacobians arise when each element v[k] of the transformed parameter vector only depends on elements u[1], ..., u[k] of the parameter vector. For triangular matrices, the determinant is the product of the diagonal elements, so the transformed parameters block of the above model can be simplified and made more efficient by recoding as follows.

```
transformed parameters {
  ...
  vector[K] J_diag;  // diagonals of Jacobian matrix
  ...
  ... compute J[k,k] = d.v[k] / d.u[k] ...
  incement_log_prob(sum(log(J_diag)));
  ...
```

18.   Custom Probability Functions

Custom distributions may also be implemented directly within Stan's programming language. The only thing that is needed is to increment the total log probability. The rest of the chapter provides two examples.

18.1.   Examples

Triangle Distribution

A simple example is the triangle distribution, whose density is shaped like an isosceles triangle with corners at specified bounds and height determined by the constraint that a density integrate to 1. If $\alpha \in \mathbb{R}$ and $\beta \in \mathbb{R}$ are the bounds, with $\alpha < \beta$, then $y \in (\alpha, \beta)$ has a density defined as follows.

$$\text{Triangle}(y|\alpha, \beta) = \frac{2}{\beta - \alpha} \left(1 - \left| y - \frac{\alpha + \beta}{\beta - \alpha} \right| \right)$$

If $\alpha = -1$, $\beta = 1$, and $y \in (-1, 1)$, this reduces to

$$\text{Triangle}(y| - 1, 1) = 1 - |y|.$$

The file `src/models/basic_distributions/triangle.stan` contains the following Stan implementation of a sampler from $\text{Triangle}(-1, 1)$.

```
parameters {
  real<lower=-1,upper=1> y;
}
model {
  increment_log_prob(log1m(fabs(y)));
}
```

The single scalar parameter y is declared as lying in the interval `(-1,1)`. The total log probability is incremented with the joint log probability of all parameters, i.e., $\log \text{Triangle}(y| - 1, 1)$. This value is coded in Stan as `log1m(fabs(y))`. The function `log1m` is is defined so that `log1m(x)` has the same value as `log(1.0-x)`, but the computation is faster, more accurate, and more stable.

    The constrained type `real<lower=-1,upper=1>` declared for y is critical for correct sampling behavior. If the constraint on y is removed from the program, say by declaring y as having the unconstrained scalar type `real`, the program would compile, but it would produce arithmetic exceptions at run time when the sampler explored values of y outside of $(-1, 1)$.

Now suppose the log probability function were extended to all of $\mathbb{R}$ as follows by defining the probability to be `log(0.0)`, i.e., $-\infty$, for values outside of $(-1, 1)$.

```
increment_log_prob(log(fmax(0.0,1 - fabs(y))));
```

With the constraint on y in place, this is just a less efficient, slower, and less arithmetically stable version of the original program. But if the constraint on y is removed, the model will compile and run without arithmetic errors, but will not sample properly.[1]

Exponential Distribution

If Stan didn't happen to include the exponential distribution, it could be coded directly using the following assignment statement, where `lambda` is the inverse scale and y the sampled variate.

```
increment_log_prob(log(lambda) - y * lambda);
```

This encoding will work for any `lambda` and y; they can be parameters, data, or one of each, or even local variables.

The assignment statement in the previous paragraph generates C++ code that is very similar to that generated by the following sampling statement.

```
y ~ exponential(lambda);
```

There are two notable differences. First, the sampling statement will check the inputs to make sure both `lambda` is positive and y is non-negative (which includes checking that neither is the special not-a-number value).

The second difference is that if `lambda` is not a parameter, transformed parameter, or local model variable, the sampling statement is clever enough to drop the `log(lambda)` term. This results in the same posterior because Stan only needs the log probability up to an additive constant. If `lambda` and y are both constants, the sampling statement will drop both terms (but still check for out-of-domain errors on the inputs).

---

[1]The problem is the (extremely!) light tails of the triangle distribution. The standard HMC and NUTS samplers can't get into the corners of the triangle properly. Because the Stan code declares y to be of type `real<lower=-1,upper=1>`, the inverse logit transform is applied to the unconstrained variable and its log absolute derivative added to the log probability. The resulting distribution on the logit-transformed y is well behaved. See Chapter 49 for more information on the transforms used by Stan.

19. Optimizing Stan Code

This chapter provides a grab bag of techniques for optimizing Stan code, including vectorization, sufficient statistics, and conjugacy.

19.1. Reparameterization

Stan's sampler can be slow in sampling from distributions with difficult posterior geometries. One way to speed up such models is through reparameterization.

Example: Neal's Funnel

In this section, we discuss a general transform from a centered to a non-centered parameterization Papaspiliopoulos et al. (2007).[1] This reparameterization is helpful because it separates the hierarchical parameters and lower-level parameters in the prior.

(Neal, 2003) defines a distribution that exemplifies the difficulties of sampling from some hierarchical models. Neal's example is fairly extreme, but can be trivially reparameterized in such a way as to make sampling straightforward.

Neal's example has support for $y \in \mathbb{R}$ and $x \in \mathbb{R}^9$ with density

$$p(y, x) = \mathsf{Normal}(y|0, 3) \times \prod_{n=1}^{9} \mathsf{Normal}(x_n|0, \exp(y/2)).$$

The probability contours are shaped like ten-dimensional funnels. The funnel's neck is particularly sharp because of the exponential function applied to y. A plot of the log marginal density of y and the first dimension x_1 is shown in Figure 19.1.

The funnel can be implemented directly in Stan as follows.

```
parameters {
  real y;
  vector[9] x;
}
model {
  y ~ normal(0,3);
  x ~ normal(0,exp(y/2));
}
```

---

[1]This parameterization came to be known on our mailing lists as the "Matt trick" after Matt Hoffman, who independently came up with it while fitting hierarchical models in Stan.

Figure 19.1: *Neal's Funnel. (Left) The marginal density of Neal's funnel for the upper-level variable y and one lower-level variable x_1 (see the text for the formula). The blue region has log density greater than -8, the yellow region density greater than -16, and the gray background a density less than -16. (Right) 4000 draws from a run of Stan's sampler with default settings. Both plots are restricted to the shown window of x_1 and y values; some draws fell outside of the displayed area as would be expected given the density. The samples are consistent with the marginal density $p(y) = \text{Normal}(y|0, 3)$, which has mean 0 and standard deviation 3.*

When the model is expressed this way, Stan has trouble sampling from the neck of the funnel, where y is small and thus x is constrained to be near 0. This is due to the fact that the density's scale changes with y, so that a step size that works well in the body will be too large for the neck and a step size that works in the neck will be very inefficient in the body.

In this particular instance, because the analytic form of the density from which samples are drawn is known, the model can be converted to the following more efficient form.

```
parameters {
  real y_raw;
  vector[9] x_raw;
}
transformed parameters {
  real y;
  vector[9] x;
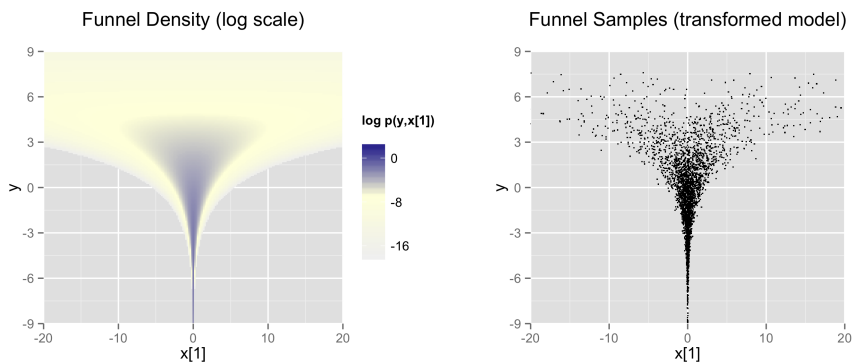
  y <- 3.0 * y_raw;
  x <- exp(y/2) * x_raw;
}
model {
  y_raw ~ normal(0,1); // implies y ~ normal(0,3)
```

```
    x_raw ~ normal(0,1); // implies x ~ normal(0,exp(y/2))
  }
```

In this second model, the parameters x_raw and y_raw are sampled as independent unit normals, which is easy for Stan. These are then transformed into samples from the funnel. In this case, the same transform may be used to define Monte Carlo samples directly based on independent unit normal samples; Markov chain Monte Carlo methods are not necessary. If such a reparameterization were used in Stan code, it is useful to provide a comment indicating what the distribution for the parameter implies for the distribution of the transformed parameter.

Reparameterizing the Cauchy

Sampling from heavy tailed distributions such as the Cauchy is difficult for Hamiltonian Monte Carlo, which operates within a Euclidean geometry.[2] The practical problem is that tail of the Cauchy requires a relatively large step size compared to the trunk. With a small step size, the No-U-Turn sampler requires many steps when starting in the tail of the distribution; with a large step size, there will be too much rejection in the central portion of the distribution. This problem may be mitigated by defining the Cauchy-distributed variable as the transform of a uniformly distributed variable using the Cauchy inverse cumulative distribution function.

Suppose a random variable of interest X has a Cauchy distribution with location μ and scale τ, so that $X \sim \mathsf{Cauchy}(\mu, \tau)$. The variable X has a cumulative distribution function $F_X : \mathbb{R} \to (0, 1)$ defined by

$$F_X(x) = \frac{1}{\pi} \arctan\left(\frac{x - \mu}{\tau}\right) + \frac{1}{2}.$$

The inverse of the cumulative distribution function, $F_X^{-1} : (0, 1) \to \mathbb{R}$, is thus

$$F_X^{-1}(y) = \mu + \tau \tan\left(\pi\left(x - \frac{1}{2}\right)\right).$$

Thus if the random variable Y has a unit uniform distribution, $Y \sim \mathsf{Uniform}(0, 1)$, then $F_X^{-1}(Y)$ has a Cauchy distribution with location μ and scale τ, i.e., $F_X^{-1}(Y) \sim \mathsf{Cauchy}(\mu, \tau)$.

Consider a Stan program involving a Cauchy-distributed parameter beta.

```
parameters {
  real beta;
```

---

[2]Riemannian Manifold Hamiltonian Monte Carlo (RMHMC) overcomes this difficulty by simulating the Hamiltonian dynamics in a a space with a position-dependent metric; see (Girolami and Calderhead, 2011) and (Betancourt, 2012).

```
    ...
  }
  model {
    beta ~ cauchy(mu,tau);
    ...
  }
```

This declaration of `beta` as a parameter may be replaced with a transformed parameter `beta` defined in terms of a uniform-distributed parameter `beta_unif`.

```
  parameters {
    real<lower=-pi()/2, upper=pi()/2> beta_unif;
    ...
  }
  transformed parameters {
    real beta;
    beta <- mu + tau * tan(beta_unif);  // beta ~ cauchy(mu,tau)
  }
  model {
    beta_unif ~ uniform(-pi()/2, pi()/2);  // not necessary
    ...
  }
```

It is more convenient in Stan to transform a uniform variable on $(-\pi/2, \pi/2)$ than one on $(0, 1)$. The Cauchy location and scale parameters, `mu` and `tau`, may be defined as data or may themselves be parameters. The variable `beta` could also be defined as a local variable if it does not need to be included in the sampler's output.

The uniform distribution on `beta_unif` is defined explicitly in the model block, but it could be safely removed from the program without changing sampling behavior. This is because $\log \text{Uniform}(\beta_{\text{unif}} | -\pi/2, \pi/2) = -\log \pi$ is a constant and Stan only needs the total log probability up to an additive constant. Stan will spend some time checking that that `beta_unif` is between `-pi()/2` and `pi()/2`, but this condition is guaranteed by the constraints in the declaration of `beta_unif`.

Reparameterizing a Student-t Distribution

One thing that sometimes works when you're having trouble with the heavy-tailedness of Student-t distributions is to use the gamma-mixture representation, which says that you can generate a Student-t distributed variable β,

$$\beta \sim \text{Student-t}(\nu, 0, 1),$$

by first generating a gamma-distributed τ,

$$\tau \sim \text{Gamma}(\nu/2, \nu/2),$$

and then generating β from the normal distribution with precision τ, which using our parameterization of the normal in terms of scale, is

$$\beta \sim \mathsf{Normal}(0, \tau^{-2}).$$

That is, the marginal distribution of β when you integrate out τ is $\mathsf{Student\text{-}t}(\nu, 0, 1)$, i.e.,

$$\mathsf{Student\text{-}t}(\beta|\nu, 0, 1). = \int_0^\infty \mathsf{Normal}(\beta|0, 1/\tau^2) \times \mathsf{Gamma}(\tau|\nu/2, \nu/2) \, d\tau.$$

You can go a step further and instead of defining a β drawn from a normal with precision τ, define α to be drawn from a unit normal,

$$\alpha \sim \mathsf{Normal}(0, 1)$$

and rescale by defining

$$\beta = \alpha/\tau^2.$$

Now suppose $\mu = \beta x$ is the product of β with a regression predictor x. Then the reparameterization $\mu = \alpha\tau^{-2}x$ has the same distribution, but in the original, direct parameterization, β has (potentially) heavy tails, whereas in the second, neither τ nor α have heavy tails.

To translate into Stan notation, this reparameterization replaces

```
parameters {
  real<lower=0> nu;
  real beta;
  ...
model {
  beta ~ student_t(nu,0,1);
  ...
```

with

```
parameters {
  real<lower=0> nu;
  real<lower=0> tau;
  ...
transformed parameters {
  real beta;
  beta <- alpha / pow(tau,2);
  ...
model {
```

```
      real half_nu;
      half_nu <- 0.5 * nu;
      tau ~ gamma(half_nu, half_nu);
      alpha ~ normal(0, 1);
      ...
```

In most cases, the lower bound for `nu` can be set to `1` or higher; when `nu` is `1`, the result is a Cauchy distribution with very fat tails and as `nu` approaches infinity, the distribution approaches a normal distribution. So the model for `nu` effectively parameterizes the heaviness of the tails of the model.

Hierarchical Models

Unfortunately, the usual situation in applied Bayesian modeling involves complex geometries and interactions that are not known analytically. Nevertheless, reparameterization can still be very effective for separating parameters. For example, a vectorized hierarchical model might draw a vector of coefficients β with definitions as follows.

```
    parameters {
      real mu_beta;
      real<lower=0> sigma_beta;
      vector[K] beta;
      ...
    model {
      beta ~ normal(mu_beta,sigma_beta);
      ...
```

Although not shown, a full model will have priors on both `mu_beta` and `sigma_beta` along with data modeled based on these coefficients. For instance, a standard binary logistic regression with data matrix `x` and binary outcome vector `y` would include a likelihood statement such as form `y ~ bernoulli_logit(x * beta)`, leading to an analytically intractable posterior.

A hierarchical model such as the above will suffer from the same kind of inefficiencies as Neal's funnel, though typically not so extreme, because the values of `beta`, `mu_beta` and `sigma_beta` are highly correlated in the posterior. Such a hierarchical model can be made much more efficient in terms of effective sample size by reparameterizing in exactly the same way as the funnel example.

```
    parameters {
      vector[K] beta_raw;
      ...
    transformed parameters {
      vector[K] beta;
```

```
      // implies: beta ~ normal(mu_beta,sigma_beta)
      beta <- mu_beta + sigma_beta * beta_raw;
    model {
      beta_raw ~ normal(0,1);
      ...
```

Any priors defined for `mu_beta` and `sigma_beta` remain as defined in the original model.

Reparameterization of hierarchical models is not limited to the normal distribution, although the normal distribution is the best candidate for doing so. In general, any distribution of parameters in the location-scale family is a good candidate for reparameterization. Let $\beta = l + s\alpha$ where l is a location parameter and s is a scale parameter. Note that l need not be the mean, s need not be the standard deviation, and neither the mean nor the standard deviation need to exist. If α and β are from the same distributional family but α has location zero and unit scale, while β has location l and scale s, then that distribution is a location-scale distribution. Thus, if α were a parameter and β were a transformed parameter, then a prior distribution from the location-scale family on α with location zero and unit scale implies a prior distribution on β with location l and scale s. Doing so would reduce the dependence between α, l, and s.

There are several univariate distributions in the location-scale family, such as the Student t distribution, including its special cases of the Cauchy distribution (with one degree of freedom) and the normal distribution (with infinite degrees of freedom). As shown above, if α is distributed standard normal, then β is distributed normal with mean $\mu = l$ and standard deviation $\sigma = s$. The logistic, the double exponential, the generalized extreme value distributions, and the stable distribution are also in the location-scale family.

Also, if z is distributed standard normal, then z^2 is distributed chi-squared with one degree of freedom. By summing the squares of K independent standard normal variates, one can obtain a single variate that is distributed chi-squared with K degrees of freedom. However, for large K, the computational gains of this reparameterization may be overwhelmed by the computational cost of specifying K primitive parameters just to obtain one transformed parameter to use in a model.

Multivariate Reparameterizations

The benefits of reparameterization are not limited to univariate distributions. A parameter with a multivariate normal prior distribution is also an excellent candidate for reparameterization. Suppose you intend the prior for β to be multivariate normal with mean vector μ and covariance matrix Σ. Such a belief is reflected by the following code.

```
data {
  int<lower=2> K;
  vector[K] mu;
  cov_matrix[K] Sigma;
  ...
parameters {
  vector[K] beta;
  ...
model {
  beta ~ multi_normal(mu,Sigma);
  ...
```

In this case mu and Sigma are fixed data, but they could be unknown parameters, in which case their priors would be unaffected by a reparameterization of beta.

If α has the same dimensions as β but the elements of α are independently and identically distributed standard normal such that $\beta = \mu + L\alpha$, where $LL^\top = \Sigma$, then β is distributed multivariate normal with mean vector μ and covariance matrix Σ. One choice for L is the Cholesky factor of Σ. Thus, the model above could be reparameterized as follows.

```
data {
  int<lower=2> K;
  vector[K] mu;
  cov_matrix[K] Sigma;
  ...
transformed data {
  matrix[K,K] L;
  L <- cholesky_decompose(Sigma);
}
parameters {
  vector[K] alpha;
  ...
transformed parameters {
  vector[K] beta;
  beta <- mu + L * alpha;
}
model {
  alpha ~ normal(0,1);
  // implies: beta ~ multi_normal(mu, Sigma)
  ...
```

This reparameterization is more efficient for two reasons. First, it reduces dependence among the elements of alpha and second, it avoids the need to invert Sigma every time multi_normal is evaluated.

The Cholesky factor is also useful when a covariance matrix is decomposed into a correlation matrix that is multiplied from both sides by a diagonal matrix of standard deviations, where either the standard deviations or the correlations are unknown parameters. The Cholesky factor of the covariance matrix is equal to the product of a diagonal matrix of standard deviations and the Cholesky factor of the correlation matrix. Furthermore, the product of a diagonal matrix of standard deviations and a vector is equal to the elementwise product between the standard deviations and that vector. Thus, if for example the correlation matrix Tau were fixed data but the vector of standard deviations sigma were unknown parameters, then a reparameterization of beta in terms of alpha could be implemented as follows.

```
data {
  int<lower=2> K;
  vector[K] mu;
  corr_matrix[K] Tau;
  ...
transformed data {
  matrix[K,K] L;
  L <- cholesky_decompose(Tau);
}
parameters {
  vector[K] alpha;
  vector<lower=0>[K] sigma;
  ...
transformed parameters {
  vector[K] beta;
  // This equals mu + diag_matrix(sigma) * L * alpha;
  beta <- mu + sigma .* (L * alpha);
}
model {
  sigma ~ cauchy(0,5);
  alpha ~ normal(0,1);
  // implies: beta ~ multi_normal(mu,
  //  diag_matrix(sigma) * L * L' * diag_matrix(sigma)))
  ...
```

This reparameterization of a multivariate normal distribution in terms of standard normal variates can be extended to other multivariate distributions that can be conceptualized as contaminations of the multivariate normal, such as the multivariate Student t and the skew multivariate normal distribution.

A Wishart distribution can also be reparameterized in terms of standard normal variates and chi-squared variates. Let L be the Cholesky factor of a $K \times K$ positive

definite scale matrix S and let ν be the degrees of freedom. If

$$
A = \begin{pmatrix}
\sqrt{c_1} & 0 & \cdots & & 0 \\
z_{21} & \sqrt{c_2} & \ddots & & \vdots \\
\vdots & \ddots & \ddots & & 0 \\
z_{K1} & \cdots & & z_{K(K-1)} & \sqrt{c_K}
\end{pmatrix},
$$

where each c_i is distributed chi-squared with $\nu - i + 1$ degrees of freedom and each z_{ij} is distributed standard normal, then $W = LAA^\top L^\top$ is distributed Wishart with scale matrix $S = LL^\top$ and degrees of freedom ν. Such a reparameterization can be implemented by the following Stan code:

```
data {
  int<lower=1> N;
  int<lower=1> K;
  int<lower=K+2> nu
  matrix[K,K] L; // Cholesky factor of scale matrix
  vector[K] mu;
  matrix[N,K] y;
  ...
parameters {
  vector<lower=0>[K] c;
  vector[0.5 * K * (K - 1)] z;
  ...
model {
  matrix[K,K] A;
  int count;
  count <- 1;
  for (j in 1:(K-1)) {
    for (i in (j+1):K) {
      A[i,j] <- z[count];
      count <- count + 1;
    }
    for (i in 1:(j - 1)) {
      A[i,j] <- 0.0;
    }
    A[j,j] <- sqrt(c[j]);
  }

  for (i in 1:K) {
    c[i] ~ chi_square(nu - i + 1);
  }
  z ~ normal(0,1);
```

```
      // implies: L * A * A' * L' ~ wishart(nu, L * L')
      y ~ multi_normal_cholesky(mu, L * A);
      ...
```

This reparameterization is more efficient for three reasons. First, it reduces dependence among the elements of z and second, it avoids the need to invert the covariance matrix, W every time wishart is evaluated. Third, if W is to be used with a multivariate normal distribution, you can pass LA to the more efficient multi_normal_cholesky function, rather than passing W to multi_normal.

If W is distributed Wishart with scale matrix S and degrees of freedom ν, then W^{-1} is distributed inverse Wishart with inverse scale matrix S^{-1} and degrees of freedom ν. Thus, the previous result can be used to reparameterize the inverse Wishart distribution. Since $W = L * A * A^\top * L^\top$, $W^{-1} = L^{\top^{-1}} A^{\top^{-1}} A^{-1} L^{-1}$, where all four inverses exist, but $L^{-1^\top} = L^{\top^{-1}}$ and $A^{-1^\top} = A^{\top^{-1}}$. We can slightly modify the above Stan code for this case:

```
data {
  int<lower=1> K;
  int<lower=K+2> nu
  matrix[K,K] L; // Cholesky factor of scale matrix
  ...
transformed data {
  matrix[K,K] eye;
  matrix[K,K] L_inv;
  for (j in 1:K) {
    for (i in 1:K) {
      eye[i,j] <- 0.0;
    }
    eye[j,j] <- 1.0;
  }
  L_inv <- mdivide_left_tri_low(L, eye);
}
parameters {
  vector<lower=0>[K] c;
  vector[0.5 * K * (K - 1)] z;
  ...
model {
  matrix[K,K] A;
  matrix[K,K] A_inv_L_inv;
  int count;
  count <- 1;
  for (j in 1:(K-1)) {
    for (i in (j+1):K) {
      A[i,j] <- z[count];
```

```
      count <- count + 1;
    }
    for (i in 1:(j - 1)) {
      A[i,j] <- 0.0;
    }
    A[j,j] <- sqrt(c[j]);
  }
  A_inv_L_inv <- mdivide_left_tri_low(A, L_inv);
  for (i in 1:K) {
    c[i] ~ chi_square(nu - i + 1);
  }
  z ~ normal(0,1); // implies: crossprod(A_inv_L_inv) ~
  // inv_wishart(nu, L_inv' * L_inv)
  ...
```

Another candidate for reparameterization is the Dirichlet distribution with all K shape parameters equal. Zyczkowski and Sommers (2001) shows that if θ_i is equal to the sum of β independent squared standard normal variates and $\rho_i = \frac{\theta_i}{\sum \theta_i}$, then the K-vector ρ is distributed Dirichlet with all shape parameters equal to $\frac{\beta}{2}$. In particular, if $\beta = 2$, then ρ is distributed uniformly on the unit simplex. Thus, we can make ρ be a transformed parameter to reduce dependence, as in:

```
data {
  int<lower=1> beta;
  ...
parameters {
  vector[beta] z[K];
  ...
transformed parameters {
  simplex[K] rho;
  for (k in 1:K)
    rho[k] <- dot_self(z[k]); // sum-of-squares
  rho <- rho / sum(rho);
}
model {
  for (k in 1:K)
    z[k] ~ normal(0,1);
  // implies: rho ~ dirichlet(0.5 * beta * ones)
  ...
```

19.2.   Vectorization

Gradient Bottleneck

Stan spends the vast majority of its time computing the gradient of the log probability function, making gradients the obvious target for optimization. Stan's gradient calculations with algorithmic differentiation require a template expression to be allocated[3] and constructed for each subexpression of a Stan program involving parameters or transformed parameters. This section defines optimization strategies based on vectorizing these subexpressions to reduce the work done during algorithmic differentiation.

Vectorizing Summations

Because of the gradient bottleneck described in the previous section, it is more efficient to collect a sequence of summands into a vector or array and then apply the `sum()` operation than it is to continually increment a variable by assignment and addition. For example, consider the following code snippet, where `foo()` is some operation that depends on `n`.

```
for (n in 1:N)
  total <- total + foo(n,...);
```

This code has to create intermediate representations for each of the N summands.

A faster alternative is to copy the values into a vector, then apply the `sum()` operator, as in the following refactoring.

```
{
  vector[N] summands;
  for (n in 1:N)
    summands[n] <- foo(n,...);
  total <- sum(summands);
}
```

Syntactically, the replacement is a statement block delineated by curly brackets ({, }), starting with the definition of the local variable `summands`.

Even though it involves extra work to allocate the `summands` vector and copy N values into it, the savings in differentiation more than make up for it. Perhaps surprisingly, it will also use substantially less memory overall than incrementing `total` within the loop.

---

[3]Stan uses its own arena-based allocation, so allocation and deallocation are faster than with a raw call to `new`.

Vectorization through Matrix Operations

The following program directly encodes a linear regression with fixed unit noise using a two-dimensional array x of predictors, an array y of outcomes, and an array `beta` of regression coefficients.

```
data {
  int<lower=1> K;
  int<lower=1> N;
  real x[K,N];
  real y[N];
}
parameters {
  real beta[K];
}
model {
  for (n in 1:N) {
    real gamma;
    gamma <- 0.0;
    for (k in 1:K)
      gamma <- gamma + x[n,k] * beta[k];
    y[n] ~ normal(gamma,1);
  }
}
```

The following model computes the same log probability function as the previous model, even supporting the same input files for data and initialization.

```
data {
  int<lower=1> K;
  int<lower=1> N;
  vector[K] x[N];
  real y[N];
}
parameters {
  vector[K] beta;
}
model {
  for (n in 1:N)
    y[n] ~ normal(dot_product(x[n],beta), 1);
}
```

Although it produces equivalent results, the dot product should not be replaced with a transpose and multiply, as in

```
        y[n] ~ normal(x[n]' * beta, 1);
```

The relative inefficiency of the transpose and multiply approach is that the transposition operator allocates a new vector into which the result of the transposition is copied. This consumes both time and memory[4]. The inefficiency of transposition could itself be mitigated somewhat by reordering the product and pulling the transposition out of the loop, as follows.

```
...
transformed parameters {
  row_vector[K] beta_t;
  beta_t <- beta';
}
model {
  for (n in 1:N)
    y[n] ~ normal(beta_t * x[n], 1);
}
```

The problem with transposition could be completely solved by directly encoding the x as a row vector, as in the following example.

```
data {
  ...
  row_vector[K] x[N];
  ...
}
parameters {
  vector[K] beta;
}
model {
  for (n in 1:N)
    y[n] ~ normal(x[n] * beta, 1);
}
```

Declaring the data as a matrix and then computing all the predictors at once using matrix multiplication is more efficient still, as in the example discussed in the next section.

Vectorized Probability Functions

The final and most efficient version replaces the loops and transformed parameters by using the vectorized form of the normal probability function, as in the following example.

---

[4]Future versions of Stan may remove this inefficiency by more fully exploiting expression templates inside the Eigen C++ matrix library. This will require enhancing Eigen to deal with mixed-type arguments, such as the type `double` used for constants and the algorithmic differentiation type `stan::agrad::var` used for variables.

```
data {
  int<lower=1> K;
  int<lower=1> N;
  matrix[N,K] x;
  vector[N] y;
}
parameters {
  vector[K] beta;
}
model {
  y ~ normal(x * beta, 1);
}
```

The variables are all declared as either matrix or vector types. The result of the matrix-vector multiplication x * beta in the model block is a vector of the same length as y.

The probability function documentation in Part V indicates which of Stan's probability functions support vectorization; see Section 26.1 for more information. Vectorized probability functions accept either vector or scalar inputs for all arguments, with the only restriction being that all vector arguments are the same dimensionality. In the example above, y is a vector of size N, x * beta is a vector of size N, and 1 is a scalar.

19.3. Exploiting Sufficient Statistics

In some cases, models can be recoded to exploit sufficient statistics in estimation. This can lead to large efficiency gains compared to an expanded model. For example, consider the following Bernoulli sampling model.

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

In this model, the sum of positive outcomes in y is a sufficient statistic for the chance of success `theta`. The model may be recoded using the binomial distribution as follows.

```
theta ~ beta(alpha,beta);
sum(y) ~ binomial(N,theta);
```

Because truth is represented as one and falsehood as zero, the sum `sum(y)` of a binary vector y is equal to the number of positive outcomes out of a total of N trials.

19.4. Exploiting Conjugacy

Continuing the model from the previous section, the conjugacy of the beta prior and binomial sampling distribution allow the model to be further optimized to the following equivalent form.

```
theta ~ beta(alpha + sum(y), beta + N - sum(y));
```

To make the model even more efficient, a transformed data variable defined to be `sum(y)` could be used in the place of `sum(y)`.

19.5. Standardizing Predictors and Outputs

Stan programs will run faster if the input is standardized to have a zero sample mean and unit sample variance. This section illustrates the principle with a simple linear regression.

Suppose that $y = (y_1, \ldots, y_N)$ is a sequence of N outcomes and $x = (x_1, \ldots, x_N)$ a parallel sequence of N predictors. A simple linear regression involving an intercept coefficient α and slope coefficient β can be expressed as

$$y_n = \alpha + \beta x_n + \epsilon_n,$$

where

$$\epsilon_n \sim \mathsf{Normal}(0, \sigma).$$

If either vector x or y has very large or very small values or if the sample mean of the values is far away from 0 (on the scale of the values), then it can be more efficient to standardize the outputs y_n and predictors x_n. The data is first centered by subtracting the sample mean, and then scaled by dividing by the sample deviation. Thus a data point u is standardized is standardized with respect to a vector y by the function z_y, defined by

$$z_y(u) = \frac{u - \bar{y}}{\mathrm{sd}(y)}$$

where the sample mean of y is

$$\bar{y} = \frac{1}{N} \sum_{n=1}^{N} y_n,$$

and the sample standard deviation of y is

$$sd(y) = \left(\frac{1}{N} \sum_{n=1}^{N} (y_n - \bar{y})^2 \right)^{1/2}.$$

The inverse transform is defined by reversing the two normalization steps, first rescaling by the same deviation and relocating by the sample mean,

$$z^{-1}(v) = sd(y)v + \bar{y}.$$

To standardize a regression problem, the predictors and outcomes are standardized. This changes the scale of the variables, and hence changes the scale of the priors. Consider the following initial model.

```
data {
  int<lower=0> N;
  vector[N] y;
  vector[N] x;
}
parameters {
  real alpha;
  real beta;
  real<lower=0> sigma;
}
model {
  // priors
  alpha ~ normal(0,10);
  beta ~ normal(0,10);
  sigma ~ cauchy(0,5);
  // likelihood
  for (n in 1:N)
    y[n] ~ normal(alpha + beta * x[n], sigma);
}
```

The data block for the standardized model is identical. The standardized predictors and outputs are defined in the transformed data block.

```
data {
  int<lower=0> N;
  vector[N] y;
```

```
    vector[N] x;
}
transformed data {
  vector[N] x_std;
  vector[N] y_std;
  x_std <- (x - mean(x)) / sd(x);
  y_std <- (y - mean(y)) / sd(y);
}
parameters {
  real alpha_std;
  real beta_std;
  real<lower=0> sigma_std;
}
model {
  alpha_std ~ normal(0,10);
  beta_std ~ normal(0,10);
  sigma_std ~ cauchy(0,5);
  for (n in 1:N)
    y_std[n] ~ normal(alpha_std + beta_std * x_std[n],
                      sigma_std);
}
```

The parameters are renamed to indicate that they aren't the "natural" parameters, but the model is otherwise identical. In particular, the fairly diffuse priors on the coefficients and error scale are the same. These could have been transformed as well, but here they are left as is, because the scales make sense as very diffuse priors for standardized data; the priors could be made more informative. For instance, because the outputs y have been standardized, the error σ should not be greater than 1, because that's the scale of the noise for predictors $\alpha = \beta = 0$.

The original regression

$$y_n = \alpha + \beta x_n + \epsilon_n$$

has been transformed to a regression on the standardized variables,

$$z_y(y_n) = \alpha' + \beta' z_x(x_n) + \epsilon'_n.$$

The original parameters can be recovered with a little algebra,

$$
\begin{aligned}
y_n &= z_y^{-1}(z_y(y_n)) \\
&= z_y^{-1}\left(\alpha' + \beta' z_x(x_n) + \epsilon'_n\right) \\
&= z_y^{-1}\left(\alpha' + \beta'\left(\frac{x_n - \bar{x}}{\text{sd}(x)}\right) + \epsilon'_n\right) \\
&= \text{sd}(y)\left(\alpha' + \beta'\left(\frac{x_n - \bar{x}}{\text{sd}(x)}\right) + \epsilon'_n\right) + \bar{y} \\
&= \left(\text{sd}(y)\left(\alpha' - \beta'\frac{\bar{x}}{\text{sd}(x)}\right) + \bar{y}\right) + \left(\beta'\frac{\text{sd}(y)}{\text{sd}(x)}\right)x_n + \text{sd}(y)\epsilon'_n,
\end{aligned}
$$

from which the original scale parameter values can be read off,

$$
\alpha = \text{sd}(y)\left(\alpha' - \beta'\frac{\bar{x}}{\text{sd}(x)}\right) + \bar{y}; \qquad \beta = \beta'\frac{\text{sd}(y)}{\text{sd}(x)}; \qquad \sigma = \text{sd}(y)\sigma'.
$$

These recovered parameter values on the original scales can be calculated within Stan using a generated quantities block following the model block,

```
generated quantities {
  real alpha;
  real beta;
  real<lower=0> sigma;
  alpha <- sd(y) * (alpha_std - beta_std * mean(x) / sd(x))
          + mean(y);
  beta <- beta_std * sd(y) / sd(x);
  sigma <- sd(y) * sigma_std;
}
```

Of course, it is inefficient to compute all of the means and standard deviations every iteration; for more efficiency, these can be calculated once and stored as transformed data. Furthermore, the model sampling statement can be easily vectorized, for instance, in the transformed model, to

```
y_std ~ normal(alpha_std + beta_std * x_std, sigma_std);
```

Part IV

Modeling Language Reference

20.   Execution of a Stan Program

This chapter provides a sketch of how a compiled Stan model is executed using sampling. Optimization shares the same data reading and initialization steps, but then does optimization rather than sampling.

This sketch is elaborated in the following chapters of this part, which cover variable declarations, expressions, statements, and blocks in more detail.

20.1.   Reading and Transforming Data

The reading and transforming data steps are the same for sampling, optimization and diagnostics.

Read Data

The first step of execution is to read data into memory. For the Stan model executable, data is read from a file in the dump format (see Chapter 6).[1] All of the variables declared in the data block will be read. If a variable cannot be read, the program will halt with a message indicating which data variable is missing.

After each variable is read, if it has a declared constraint, the constraint is validated. For example, if a variable N is declared as int<lower=0>, after N is read, it will be tested to make sure it is greater than or equal to zero. If a variable violates its declared constraint, the program will halt with a warning message indicating which variable contains an illegal value, the value that was read, and the constraint that was declared.

Define Transformed Data

After data is read into the model, the transformed data variable statements are executed in order to define the transformed data variables. As the statements execute, declared constraints on variables are not enforced.

After the statements are executed, all declared constraints on transformed data variables are validated. If the validation fails, execution halts and the variable's name, value and constraints are displayed.

---

[1]The C++ code underlying Stan is flexible enough to allow data to be read from memory or file. Calls from R, for instance, can be configured to read data from file or directly from R's memory.

20.2. Initialization

Initialization is the same for sampling, optimization, and diagnosis

User-Supplied Initial Values

If there are user-supplied initial values for parameters, these are read using the same input mechanism and same file format as data reads. Any constraints declared on the parameters are validated for the initial values. If a variable's value violates its declared constraint, the program halts and a diagnostic message is printed.

After being read, initial values are transformed to unconstrained values that will be used to initialize the sampler.

Random Initial Values

If there are no user-supplied initial values, the default initialization strategy is to initialize the unconstrained parameters directly with values drawn uniformly from the interval $(-2, 2)$. The bounds of this initialization can be changed but it is always symmetric around 0. The value of 0 is special in that it represents the median of the initialization. An unconstrained value of 0 corresponds to different parameter values depending on the constraints declared on the parameters.

An unconstrained real does not involve any transform, so an initial value of 0 for the unconstrained parameters is also a value of 0 for the constrained parameters.

For parameters that are bounded below at 0, the initial value of 0 on the unconstrained scale corresponds to $\exp(0) = 1$ on the constrained scale. A value of -2 corresponds to $\exp(-2) = .13$ and a value of 2 corresponds to $\exp(2) = 7.4$.

For parameters bounded above and below, the initial value of 0 on the unconstrained scale corresponds to a value at the midpoint of the constraint interval. For probability parameters, bounded below by 0 and above by 1, the transform is the inverse logit, so that an initial unconstrained value of 0 corresonds to a constrained value of 0.5, -2 corresponds to 0.12 and 2 to 0.88. Bounds other than 0 and 1 are just scaled and translated.

Simplexes with initial values of 0 on the unconstrained basis correspond to symmetric values on the constrained values (i.e., each value is $1/K$ in a K-simplex).

Cholesky factors for positive-definite matrices are initialized to 1 on the diagonal and 0 elsewhere; this is because the diagonal is log transformed and the below-diagonal values are unconstrained.

The initial values for other parameters can be determined from the transfrom that is applied. The transforms are all described in full detail in Chapter 49.

Zero Initial Values

The initial values may all be set to 0 on the unconstrained scale. This can be helpful for diagnosis, and may also be a good starting point for sampling. Once a model is running, multiple chains with more diffuse starting points can help diagnose problems with convergence; see Section 48.3 for more information on convergence monitoring.

20.3. Sampling

Sampling is based on simulating the Hamiltonian of a particle with a starting position equal to the current parameter values and an initial momentum (kinetic energy) generated randomly. The potential energy at work on the particle is taken to be the negative log (unnormalized) total probability function defined by the model. In the usual approach to implementing HMC, the Hamiltonian dynamics of the particle is simulated using the leapfrog integrator, which discretizes the smooth path of the particle into a number of small time steps called leapfrog steps.

Leapfrog Steps

For each leapfrog step, the negative log probability function and its gradient need to be evaluated at the position corresponding to the current parameter values (a more detailed sketch is provided in the next section). These are used to update the momentum based on the gradient and the position based on the momentum.

For simple models, only a few leapfrog steps with large step sizes are needed. For models with complex posterior geometries, many small leapfrog steps may be needed to accurately model the path of the parameters.

If the user specifies the number of leapfrog steps (i.e., chooses to use standard HMC), that number of leapfrog steps are simulated. If the user has not specified the number of leapfrog steps, the No-U-Turn sampler (NUTS) will determine the number of leapfrog steps adaptively (Hoffman and Gelman, 2011, 2013).

Log Probability and Gradient Calculation

During each leapfrog step, the log probability function and its gradient must be calculated. This is where most of the time in the Stan algorithm is spent. This log probability function, which is used by the sampling algorithm, is defined over the unconstrained parameters.

The first step of the calculation requires the inverse transform of the unconstrained parameter values back to the constrained parameters in terms of which the

model is defined. There is no error checking required because the inverse transform is a total function on every point in whose range satisfies the constraints.

Because the probability statements in the model are defined in terms of constrained parameters, the log Jacobian of the inverse transform must be added to the accumulated log probability.

Next, the transformed parameter statements are executed. After they complete, any constraints declared for the transformed parameters are checked. If the constraints are violated, the model will halt with a diagnostic error message.

The final step in the log probability function calculation is to execute the statements defined in the model block.

As the log probability function executes, it accumulates an in-memory representation of the expression tree used to calculate the log probability. This includes all of the transformed parameter operations and all of the Jacobian adjustments. This tree is then used to evaluate the gradients by propagating partial derivatives backward along the expression graph. The gradient calculations account for the majority of the cycles consumed by a Stan program.

Metropolis Accept/Reject

A standard Metropolis accept/reject step is required to retain detailed balance and ensure samples are marginally distributed according to the probability function defined by the model. This Metropolis adjustment is based on comparing log probabilities, here defined by the Hamiltonian, which is the sum of the potential (negative log probability) and kinetic (squared momentum) energies. In theory, the Hamiltonian is invariant over the path of the particle and rejection should never occur. In practice, the probability of rejection is determined by the accuracy of the leapfrog approximation to the true trajectory of the parameters.

If step sizes are small, very few updates will be rejected, but many steps will be required to move the same distance. If step sizes are large, more updates will be rejected, but fewer steps will be required to move the same distance. Thus a balance between effort and rejection rate is required. If the user has not specified a step size, Stan will tune the step size during warmup sampling to achieve a desired rejection rate (thus balancing rejection versus number of steps).

If the proposal is accepted, the parameters are updated to their new values. Otherwise, the sample is the current set of parameter values.

20.4. Optimization

Optimization runs very much like sampling in that it starts by reading the data and then initializing parameters. Unlike sampling, it produces a deterministic output

which requires no further analysis other than to verify that the optimizer itself converged to a posterior mode. The output for optimization is also similar to that for sampling.

20.5. Model Diagnostics

Model diagnostics are like sampling and optimization in that they depend on a model's data being read and its parameters being initialized. The output, at least so far, is just to the command line. See Section 4.2 for information on the available diagnostics (as of Stan 2.0, just gradients and log probabilities at the initialization state).

20.6. Output

For each final sample (not counting samples during warmup or samples that are thinned), there is an output stage of writing the samples.

Generated Quantities

Before generating any output, the statements in the generated quantities block are executed. This can be used for any forward simulation based on parameters of the model. Or it may be used to transform parameters to an appropriate form for output.

After the generated quantities statements execute, the constraints declared on generated quantities variables are validated. If these constraints are violated, the program will terminate with a diagnostic message.

Write

The final step is to write the actual values. The values of all variables declared as parameters, transformed parameters, or generated quantities are written. Local variables are not written, nor is the data or transformed data. All values are written in their constrained forms, that is the form that is used in the model definitions.

In the executable form of a Stan models, parameters, transformed parameters, and generated quantities are written to a file in comma-separated value (csv) notation with a header defining the names of the parameters (including indices for multivariate parameters).[2]

---

[2]In the R version of Stan, the values may either be written to a csv file or directly back to R's memory.

21. Data Types and Variable Declarations

This chapter covers the data types for expressions in Stan. Every variable used in a Stan program must have a declared data type. Only values of that type will be assignable to the variable (except for temporary states of transformed data and transformed parameter values). This follows the convention of programming languages like C++, not the conventions of scripting languages like Python or statistical languages such as R or BUGS.

The motivation for strong, static typing is threefold.

- Strong typing forces the programmer's intent to be declared with the variable, making programs easier to comprehend and hence easier to debug and maintain.

- Strong typing allows programming errors relative to the declared intent to be caught sooner (at compile time) rather than later (at run time). The Stan compiler (see Section 3.3) will flag any type errors and indicate the offending expressions quickly when the program is compiled.

- Constrained types will catch runtime data, initialization, and intermediate value errors as soon as they occur rather than allowing them to propagate and potentially pollute final results.

Strong typing disallows assigning the same variable to objects of different types at different points in the program or in different invocations of the program.

21.1. Overview of Data Types

Basic Data Types

The primitive Stan data types are `real` for continuous scalar quantities and `int` for integer values. The compound data types include `vector` (of real values), `row_vector` (of real values), and `matrix` (of real values).

Constrained Data Types

Integer or real types may be constrained with lower bounds, upper bounds, or both. There are four constrained vector data types, `simplex` for unit simplexes, `unit_vector` for unit-length vectors, `ordered` for ordered vectors of scalars and `positive_ordered` for vectors of positive ordered scalars. There are specialized matrix data types `corr_matrix` and `cov_matrix` for correlation matrices (symmetric,

positive definite, unit diagonal) and covariance matrices (symmetric, positive definite). The type `cholesky_factor_cov` is for Cholesky factors of covariance matrices (lower triangular, positive diagonal, product with own transpose is a covariance matrix).

Arrays

Stan supports arrays of arbitrary order of any of the basic data types or constrained basic data types. This includes three-dimensional arrays of integers, one-dimensional arrays of positive reals, four-dimensional arrays of simplexes, one-dimensional arrays of row vectors, and so on.

21.2.   Primitive Numerical Data Types

Unfortunately, the lovely mathematical abstraction of integers and real numbers is only partially supported by finite-precision computer arithmetic.

Integers

Stan uses 32-bit (4-byte) integers for all of its integer representations. The maximum value that can be represented as an integer is $2^{31} - 1$; the minimum value is $-(2^{31})$.

When integers overflow, their values wrap. Thus it is up to the Stan programmer to make sure the integer values in their programs stay in range. In particular, every intermediate expression must have an integer value that is in range.

Integer arithmetic works in the expected way for addition, subtraction, and multiplication, but rounds the result of division (see Section 28.1 for more information).

Reals

Stan uses 64-bit (8-byte) floating point representations of real numbers. Stan roughly[1] follows the IEEE 754 standard for floating-point computation. The range of a 64-bit number is roughly $\pm 2^{1022}$, which is slightly larger than $\pm 10^{307}$. It is a good idea to stay well away from such extreme values in Stan models as they are prone to cause overflow.

64-bit floating point representations have roughly 15 decimal digits of accuracy. But when they are combined, the result often has less accuracy. In some cases, the difference in accuracy between two operands and their result is large.

There are three special real values used to represent (1) error conditions, (2) positive infinity, and (3) negative infinity. The error value is referred to as "not a number."

---

[1]Stan compiles integers to `int` and reals to `double` types in C++. Precise details of rounding will depend on the compiler and hardware architecture on which the code is run.

Promoting Integers to Reals

Stan automatically promotes integer values to real values if necessary, but does not automatically demote real values to integers. For very large integers, this will cause a rounding error to fewer significant digits in the floating point representation than in the integer representation.

Unlike in C++, real values are never demoted to integers. Therefore, real values may only be assigned to real variables. Integer values may be assigned to either integer variables or real variables. Internally, the integer representation is cast to a floating-point representation. This operation is not without overhead and should thus be avoided where possible.

21.3. Univariate Data Types and Variable Declarations

All variables used in a Stan program must have an explicitly declared data type. The form of a declaration includes the type and the name of a variable. This section covers univariate types, the next section vector and matrix types, and the following section array types.

Unconstrained Integer

Unconstrained integers are declared using the `int` keyword. For example, the variable `N` is declared to be an integer as follows.

```
int N;
```

Constrained Integer

Integer data types may be constrained to allow values only in a specified interval by providing a lower bound, an upper bound, or both. For instance, to declare `N` to be a positive integer, use the following.

```
int<lower=1> N;
```

This illustrates that the bounds are inclusive for integers.

To declare an integer variable `cond` to take only binary values, that is zero or one, a lower and upper bound must be provided, as in the following example.

```
int<lower=0,upper=1> cond;
```

Unconstrained Real

Unconstrained real variables are declared using the keyword `real`, The following example declares `theta` to be an unconstrained continuous value.

```
real theta;
```

Constrained Real

Real variables may be bounded using the same syntax as integers. In theory (that is, with arbitrary-precision arithmetic), the bounds on real values would be exclusive. Unfortunately, finite-precision arithmetic rounding errors will often lead to values on the boundaries, so they are allowed in Stan.

The variable `sigma` may be declared to be non-negative as follows.

```
real<lower=0> sigma;
```

The following declares the variable `x` to be less than or equal to -1.

```
real<upper=-1> x;
```

To ensure `rho` takes on values between -1 and 1, use the following declaration.

```
real<lower=-1,upper=1> rho;
```

Infinite Constraints

Lower bounds that are negative infinity or upper bounds that are positive infinity are ignored. Stan provides constants `positive_infinity()` and `negative_infinity()` which may be used for this purpose, or they may be read as data in the dump format.

Expressions as Bounds

Bounds for integer or real variables may be arbitrary expressions. The only requirement is that they only include variables that have been defined before the declaration. If the bounds themselves are parameters, the behind-the-scenes variable transform accounts for them in the log Jacobian.

For example, it is acceptable to have the following declarations.

```
data {
 real lb;
}
parameters {
   real<lower=lb> phi;
}
```

This declares a real-valued parameter `phi` to take values greater than the value of the real-valued data variable `lb`. Constraints may be complex expressions, but must be of type `int` for integer variables and of type `real` for real variables (including constraints on vectors, row vectors, and matrices). Variables used in constraints can be any variable that has been defined at the point the constraint is used. For instance,

```
data {
    int<lower=1> N;
    real y[N];
}
parameters {
    real<lower=min(y),upper=max(y)> phi;
}
```

This declares a positive integer data variable `N`, an array `y` of real-valued data of length `N`, and then a parameter ranging between the minimum and maximum value of `y`. The functions `fmin()` and `fmax()` are minimum and maximum functions for floating point quantities.

21.4. Vector and Matrix Data Types

Values

Vectors, row vectors, and matrices contain real values. Arrays, on the other hand, may contain any kind of value, including integers and structured values like vectors.

Indexing

Vectors and matrices, as well as arrays, are indexed starting from one in Stan. This follows the convention in statistics and linear algebra as well as their implementations in the statistical software packages R, MATLAB, BUGS, and JAGS. General computer programming languages, on the other hand, such as C++ and Python, index arrays starting from zero.

Vectors

Vectors in Stan are column vectors; see the next subsection for information on row vectors. Vectors are declared with a size (i.e., a dimensionality). For example, a 3-dimensional vector is declared with the keyword `vector`, as follows.

```
vector[3] u;
```

Vectors may also be declared with constraints, as in the following declaration of a 3-vector of non-negative values.

```
vector<lower=0>[3] u;
```

Unit Simplexes

A unit simplex is a vector with non-negative values whose entries sum to 1. For instance, $(0.2, 0.3, 0.4, 0.1)^\top$ is a unit 4-simplex. Unit simplexes are most often used as parameters in categorical or multinomial distributions, and they are also the sampled variate in a Dirichlet distribution. Simplexes are declared with their full dimensionality. For instance, `theta` is declared to be a unit 5-simplex by

```
simplex[5] theta;
```

Unit simplexes are implemented as vectors and may be assigned to other vectors and vice-versa. Simplex variables, like other constrained variables, are validated to ensure they contain simplex values; for simplexes, this is only done up to a statically specified accuracy threshold ϵ to account for errors arising from floating-point imprecision.

Unit Vectors

A unit vector is a vector with a norm of one. For instance, $(0.5, 0.5, 0.5, 0.5)^\top$ is a unit 4-vector. Unit vectors are sometimes used in directional statistics. Unit vectors are declared with their full dimensionality. For instance, `theta` is declared to be a unit 5-vector by

```
unit_vector[5] theta;
```

Unit vectors are implemented as vectors and may be assigned to other vectors and vice-versa. Unit vector variables, like other constrained variables, are validated to ensure that they are indeed unit length; for unit vectors, this is only done up to a statically specified accuracy threshold ϵ to account for errors arising from floating-point imprecision.

Ordered Vectors

An ordered vector type in Stan represents a vector whose entries are sorted in ascending order. For instance, $(-1.3, 2.7, 2.71)^\top$ is an ordered 3-vector. Ordered vectors are most often employed as cut points in ordered logistic regression models (see Section 12.6).

The variable `c` is declared as an ordered 5-vector by

```
ordered[5] c;
```

After their declaration, ordered vectors, like unit simplexes, may be assigned to other vectors and other vectors may be assigned to them. Constraints will be checked after executing the block in which the variables were declared.

Positive, Ordered Vectors

There is also a positive, ordered vector type which operates similarly to ordered vectors, but all entries are constrained to be positive. For instance, $(2, 3.7, 4, 12.9)$ is a positive, ordered 4-vector.

The variable d is declared as a positive, ordered 5-vector by

```
positive_ordered[5] d;
```

Like ordered vectors, after their declaration positive ordered vectors assigned to other vectors and other vectors may be assigned to them. Constraints will be checked after executing the block in which the variables were declared.

Row Vectors

Row vectors are declared with the keyword row_vector. Like (column) vectors, they are declared with a size. For example, a 1093-dimensional row vector u would be declared as

```
row_vector[1093] u;
```

Constraints are declared as for vectors, as in the following example of a 10-vector with values between -1 and 1.

```
row_vector<lower=-1,upper=1>[10] u;
```

Row vectors may not be assigned to column vectors, nor may column vectors be assigned to row vectors. If assignments are required, they may be accommodated through the transposition operator.

Matrices

Matrices are declared with the keyword matrix along with a number of rows and number of columns. For example,

```
matrix[3,3] A;
matrix[M,N] B;
```

declares A to be a 3×3 matrix and B to be a $M \times N$ matrix. For the second declaration to be well formed, the variables M and N must be declared as integers in either the data or transformed data block and before the matrix declaration.

Matrices may also be declared with constraints, as in this (3×4) matrix of non-positive values.

```
matrix<upper=0>[3,4] B;
```

Assigning to Rows of a Matrix

Rows of a matrix can be assigned by indexing the left-hand side of an assignment statement. For example, this is possible.

```
matrix[M,N] a;
row_vector[N] b;
...
a[1] <- b;
```

This copies the values from row vector b to a[1], which is the first row of the matrix a. If the number of columns in a is not the same as the size of b, a run-time error is raised; the number of rows of a is N, which is also the size of b.

Assignment works by copying values in Stan. That means any subsequent assignment to a[1] does not affect b, nor does an assignment to b affect a.

Correlation Matrices

Matrix variables may be constrained to represent correlation matrices. A matrix is a correlation matrix if it is symmetric and positive definite, has entries between -1 and 1, and has a unit diagonal. Because correlation matrices are square, only one dimension needs to be declared. For example,

```
corr_matrix[3] Sigma;
```

declares Sigma to be a 3×3 correlation matrix.

Correlation matrices may be assigned to other matrices, including unconstrained matrices, if their dimensions match, and vice-versa.

Covariance Matrices

Matrix variables may be constrained to represent covariance matrices. A matrix is a covariance matrix if it is symmetric and positive definite. Like correlation matrices, covariance matrices only need a single dimension in their declaration. For instance,

```
    cov_matrix[K] Omega;
```

declares `Omega` to be a $K \times K$ covariance matrix, where K is the value of the data variable K.

Cholesky Factors of Covariance Matrices

Matrix variables may be constrained to represent the Cholesky factors of a covariance matrix. This is often more convenient or more efficient than representing covariance matrices directly.

A Cholesky factor L is an $M \times N$ lower-triangular matrix (if $m < n$ then $L[m, n] = 0$) with a positive diagonal ($L[k, k] = 0$) and $M \geq N$. If L is a Cholesky factor, then $\Sigma = L L^\top$ is a covariance matrix. Furthermore, every covariance matrix has a Cholesky factorization.

The typical case of a square Cholesky factor may be declared with a single dimension,

```
    cholesky_factor_cov[4] L;
```

In general, two dimensions may be declared, with the above being equal to `cholesky_factor_cov[4,4]`. The type `cholesky_factor_cov[M,N]` may be used for the general $M \times N$.

Assigning Constrained Variables

Constrained variables of all types may be assigned to other variables of the same unconstrained type and vice-versa. For instance, a variable declared to be `real<lower=0,upper=1>` could be assigned to a variable declared as `real` and vice-versa. Similarly, a variable declared as `matrix[3,3]` may be assigned to a variable declared as `cov_matrix[3]` or `cholesky_factor_cov[3]`, and vice-versa.

Checks are carried out at the end of each relevant block of statements to ensure constraints are enforced. This includes run-time size checks. The Stan compiler isn't able to catch the fact that an attempt may be made to assign a matrix of one dimensionality to a matrix of mismatching dimensionality.

Expressions as Size Declarations

Variables may be declared with sizes given by expressions. Such expressions are constrained to only contain data or transformed data variables. This ensures that all sizes are determined once the data is read in and transformed data variables defined by their statements. For example, the following is legal.

```
data {
  int<lower=0> N_observed;    int<lower=0> N_missing;
  ...
transformed parameters {
  vector[N_observed + N_missing] y;
  ...
```

Accessing Vector and Matrix Elements

If `v` is a column vector or row vector, then `v[2]` is the second element in the vector. If `m` is a matrix, then `m[2,3]` is the value in the second row and third column.

Providing a matrix with a single index returns the specified row. For instance, if `m` is a matrix, then `m[2]` is the second row. This allows Stan blocks such as

```
matrix[M,N] m;
row_vector[N] v;
real x;
...
v <- m[2];
x <- v[3];    // x == m[2][3] == m[2,3]
```

The type of `m[2]` is `row_vector` because it is the second row of `m`. Thus it is possible to write `m[2][3]` instead of `m[2,3]` to access the third element in the second row. When given a choice, the form `m[2,3]` is preferred.[2]

Size Declaration Restrictions

An integer expression is used to pick out the sizes of vectors, matrices, and arrays. For instance, we can declare a vector of size `M + N` using

```
vector[M + N] y;
```

Any integer-denoting expression may be used for the size declaration, providing all variables involved are either data, transformed data, or local variables. That is, expressions used for size declarations may not include parameters or transformed parameters or generated quantities.

---

[2]As of Stan version 1.0, the form `m[2,3]` is more efficient because it does not require the creation and use of an intermediate expression template for `m[2]`. In later versions, explicit calls to `m[2][3]` may be optimized to be as efficient as `m[2,3]` by the Stan compiler.

21.5. Array Data Types

Stan supports arrays of arbitrary dimension. An array's elements may be any of the basic data types, that is univariate integers, univariate reals, vectors, row vectors matrices, including all of the constrained forms.

Declaring Array Variables

Arrays are declared by enclosing the dimensions in square brackets following the name of the variable.

The variable n is declared as an array of five integers as follows.

```
int n[5];
```

A two-dimensional array of real values with three rows and four columns is declared with the following.

```
real a[3,4];
```

A three-dimensional array z of positive reals with five rows, four columns, and two shelves can be declared as follows.

```
real<lower=0> z[5,4,2];
```

Arrays may also be declared to contain vectors. For example,

```
vector[7] mu[3];
```

declares mu to be a 3-dimensional array of 7-vectors. Arrays may also contain matrices. The example

```
matrix[7,2] mu[15,12];
```

declares a 15×12-dimensional array of 7×2 matrices. Any of the constrained types may also be used in arrays, as in the declaration

```
cholesky_factor_cov[5,6] mu[2,3,4];
```

of a $2 \times 3 \times 4$ array of 5×6 Cholesky factors of covariance matrices.

Accessing Array Elements and Subarrays

If x is a 1-dimensional array of length 5, then x[1] is the first element in the array and x[5] is the last. For a 3×4 array y of two dimensions, y[1,1] is the first element and y[3,4] the last element. For a three-dimensional array z, the first element is z[1,1,1], and so on.

Subarrays of arrays may be accessed by providing fewer than the full number of indexes. For example, suppose y is a two-dimensional array with three rows and four columns. Then y[3] is one-dimensional array of length four. This means that y[3][1] may be used instead of y[3,1] to access the value of the first column of the third row of y. The form y[3,1] is the preferred form (see Footnote 2 in this chapter).

Subarrays may be manipulated and assigned just like any other variables. Similar to the behavior of matrices, Stan allows blocks such as

```
real w[9,10,11];
real x[10,11];
real y[11];
real z;
...
x <- w[5];
y <- x[4];  // y == w[5][4]  == w[5,4]
z <- y[3];  // z == w[5][4][3] == w[5,4,3]
```

Assigning

—fixme—

Arrays of Matrices and Vectors

Arrays of vectors and matrices are accessed in the same way as arrays of doubles. Consider the following vector and scalar declarations.

```
vector[5] a[4,3];
vector[5] b[4];
vector[5] c;
real x;
```

With these declarations, the following assignments are legal.

```
b <- a[1];     // result is array of vectors
c <- a[1,3];   // result is vector
c <- b[3];     //   same result as above
```

```
x <- a[1,3,5];  // result is scalar
x <- b[3,5];    //   same result as above
x <- c[5];      //   same result as above
```

Row vectors and other derived vector types (simplex and ordered) behave the same way in terms of indexing.

Consider the following matrix, vector and scalar declarations.

```
matrix[6,5] d[3,4];
matrix[6,5] e[4];
matrix[6,5] f;
row_vector[5] g;
real x;
```

With these declarations, the following definitions are legal.

```
e <- d[1];       // result is array of matrices
f <- d[1,3];     // result is matrix
f <- e[3];       //   same result as above
g <- d[1,3,2];   // result is row vector
g <- e[3,2];     //   same result as above
g <- f[2];       //   same result as above
x <- d[1,3,5,2]; // result is scalar
x <- e[3,5,2];   //   same result as above
x <- f[5,2];     //   same result as above
x <- g[2];       //   same result as above
```

As shown, the result `f[2]` of supplying a single index to a matrix is the indexed row, here row 2 of matrix `f`.

Partial Array Assignment

Subarrays of arrays may be assigned by indexing on the left-hand side of an assignment statement. For example, the following is legal.

```
real x[I,J,K];
real y[J,K];
real z[K];
...
x[1] <- y;
x[1,1] <- z;
```

The sizes must match. Here, x[1] is a J by K array, as is is y.

Partial array assignment also works for arrays of matrices, vectors, and row vectors.

Mixing Array, Vector, and Matrix Types

Arrays, row vectors, column vectors and matrices are not interchangeable in Stan. Thus a variable of any one of these fundamental types is not assignable to any of the others, nor may it be used as an argument where the other is required (use as arguments follows the assignment rules).

Mixing Vectors and Arrays

For example, vectors cannot be assigned to arrays or vice-versa.

```
real a[4];
vector b[4];
row_vector c[4];
...
a <- b; // illegal assignment of vector to array
b <- a; // illegal assignment of array to vector
a <- c; // illegal assignment of row vector to array
c <- a; // illegal assignment of array to row vector
```

Mixing Row and Column Vectors

It is not even legal to assign row vectors to column vectors or vice versa.

```
vector b[4];
row_vector c[4];
...
b <- c; // illegal assignment of row vector to column vector
c <- b; // illegal assignment of column vector to row vector
```

Mixing Matrices and Arrays

The same holds for matrices, where 2-dimensional arrays may not be assigned to matrices or vice-versa.

```
real a[3,4];
matrix[3,4] b;
...
```

```
    a <- b;  // illegal assignment of matrix to array
    b <- a;  // illegal assignment of array to matrix
```

Mixing Matrices and Vectors

A $1 \times N$ matrix cannot be assigned a row vector or vice versa.

```
    matrix[1,4] a;
    row_vector[4] b;
    ...
    a <- b;  // illegal assignment of row vector to matrix
    b <- a;  // illegal assignment of matrix to row vector
```

Similarly, an $M \times 1$ matrix may not be assigned to a column vector.

```
    matrix[4,1] a;
    vector[4] b;
    ...
    a <- b;  // illegal assignment of column vector to matrix
    b <- a;  // illegal assignment of matrix to column vector
```

Size Declaration Restrictions

An integer expression is used to pick out the sizes of arrays. The same restrictions as for vector and matrix sizes apply, namely that the size is declared with an integer-denoting expression that does not contain any parameters, transformed parameters, or generated quantities.

21.6. Variable Types vs. Constraints and Sizes

The type information associated with a variable only contains the underlying type and dimensionality of the variable.

Type Information Excludes Sizes

The size associated with a given variable is not part of its data type. For example, declaring a variable using

```
    real a[3];
```

declares the variable a to be an array. The fact that it was declared to have size 3 is part of its declaration, but not part of its underlying type.

When are Sizes Checked?

Sizes are determined dynamically (at run time) and thus cannot be type-checked statically when the program is compiled. As a result, any conformance error on size will raise a run-time error. For example, trying to assign an array of size 5 to an array of size 6 will cause a run-time error. Similarly, multiplying an $N \times M$ by a $J \times K$ matrix will raise a run-time error if $M \neq J$.

Type Information Excludes Constraints

Like sizes, constraints are not treated as part of a variable's type in Stan when it comes to the compile-time check of operations it may participate in. Anywhere Stan accepts a matrix as an argument, it will syntactically accept a correlation matrix or covariance matrix or Cholesky factor. Thus a covariance matrix may be assigned to a matrix and vice-versa.

Similarly, a bounded real may be assigned to an unconstrained real and vice-versa.

When are Function Argument Constraints Checked?

For arguments to functions, constraints are sometimes, but not always checked when the function is called. Exclusions include C++ standard library functions. All probabiliy functions and cumulative distribution functions check that their arguments are appropriate at run time as the function is called.

When are Declared Variable Constraints Checked?

For data variables, constraints are checked after the variable is read from a data file or other source. For transformed data variables, the check is done after the statements in the transformed data block have executed. Thus it is legal for intermediate values of variables to not satisfy declared constraints.

For parameters, constraints are enforced by the transform applied and do not need to be checked. For transformed parameters, the check is done after the statements in the transformed parameter block have executed.

For generated quantities, constraints are enforced after the statements in the generated quantities block have executed.

Type Naming Notation

In order to refer to data types, it is convenient to have a way to refer to them. The type naming notation outlined in this section is not part of the Stan programming language, but rather a convention adopted in this document to enable a concise description of a type.

Because size information is not part of a data type, data types will be written without size information. For instance, `real[]` is the type of one-dimensional array of reals and `matrix` is the type of matrices. The three-dimensional integer array type is written as `int[ ,  ,]`, indicating the number slots available for indexing. Similarly, `vector[,]` is the type of a two-dimensional array of vectors.

22. Expressions

An expression is the basic syntactic unit in a Stan program that denotes a value. Every expression in a well-formed Stan program has a type that is determined statically (at compile time). If an expressions type cannot be determined statically, the Stan compiler (see Section 3.3) will report the location of the problem.

This chapter covers the syntax, typing, and usage of the various forms of expressions in Stan.

22.1. Numeric Literals

The simplest form of expression is a literal that denotes a primitive numerical value.

Integer Literals

Integer literals represent integers of type `int`. Integer literals are written in base 10 without any separators. Integer literals may contain a single negative sign. (The expression `--1` is interpreted as the negation of the literal `-1`.)

The following list contains well-formed integer literals.

    0, 1, -1, 256, -127098, 24567898765

Integer literals must have values that fall within the bounds for integer values (see Section 21.2).

Integer literals may not contain decimal points (`.`). Thus the expressions `1.` and `1.0` are of type `real` and may not be used where a value of type `int` is required.

Real Literals

A number written with a period or with scientific notation is assigned to a the continuous numeric type `real`. Real literals are written in base 10 with a period (`.`) as a separator. Examples of well-formed real literals include the following.

    0.0, 1.0, 3.14, -217.9387, 2.7e3, -2E-5

The notation `e` or `E` followed by a positive or negative integer denotes a power of 10 to multiply. For instance, `2.7e3` denotes 2.7×10^3 and `-2E-5` denotes -2×10^{-5}.

22.2. Variables

A variable by itself is a well-formed expression of the same type as the variable. Variables in Stan consist of ASCII strings containing only the basic lower-case and upper-case Roman letters, digits, and the underscore (_) character. Variables must start with a letter (a–z and A–Z) and may not end with two underscores (__).

Examples of legal variable identifiers are as follows.

```
a, a3, a_3, Sigma, my_cpp_style_variable, myCamelCaseVariable
```

Unlike in R and BUGS, variable identifiers in Stan may not contain a period character.

Reserved Names

Stan reserves many strings for internal use and these may not be used as the name of a variable. An attempt to name a variable after an internal string results in the `stanc` translator halting with an error message indicating which reserved name was used and its location in the model code.

Model Name

The name of the model cannot be used as a variable within the model. This is usually not a problem because the default in `bin/stanc` is to append `_model` to the name of the file containing the model specification. For example, if the model is in file `foo.stan`, it would not be legal to have a variable named `foo_model` when using the default model name through `bin/stanc`. With user-specified model names, variables cannot match the model.

Reserved Words from Stan Language

The following list contains reserved words for Stan's programming language. Not all of these features are implemented in Stan yet, but the tokens are reserved for future use.

```
for, in, while, repeat, until, if, then, else, true, false
```

Variables should not be named after types, either, and thus may not be any of the following.

```
int,   real,   vector,   simplex,   unit_vector,   ordered,
positive_ordered,   row_vector,   matrix,   cholesky_factor_cov,
corr_matrix, cov_matrix.
```

Variable names will *not* conflict with the following block identifiers,

```
model, data, parameters, quantities, transformed, generated,
```

Reserved Names from Stan Implementation

Some variable names are reserved because they are used within Stan's C++ implementation. These are

> var fvar

Reserved Function and Distribution Names

Variable names will conflict with the names of predefined functions other than constants. Thus a variable may not be named `logit` or `add`, but it may be named `pi` or `e`.

Variable names will also conflict with the names of distributions suffixed with `_log`, `_cdf`, `_cdf_log`, and `_ccdf_log`, such as `normal_cdf_log`.

Using any of these variable names causes the `stanc` translator to halt and report the name and location of the variable causing the conflict.

Reserved Names from C++

Finally, variable names, including the names of models, should not conflict with any of the C++ keywords.

> `alignas, alignof, and, and_eq, asm, auto, bitand, bitor, bool,`
> `break, case, catch, char, char16_t, char32_t, class, compl,`
> `const, constexpr, const_cast, continue, decltype, default, delete,`
> `do, double, dynamic_cast, else, enum, explicit, export, extern,`
> `false, float, for, friend, goto, if, inline, int, long, mutable,`
> `namespace, new, noexcept, not, not_eq, nullptr, operator, or, or_eq,`
> `private, protected, public, register, reinterpret_cast, return,`
> `short, signed, sizeof, static, static_assert, static_cast, struct,`
> `switch, template, this, thread_local, throw, true, try, typedef,`
> `typeid, typename, union, unsigned, using, virtual, void, volatile,`
> `wchar_t, while, xor, xor_eq`

Legal Characters

The legal variable characters have the same ASCII code points in the range 0–127 as in Unicode.

Characters	ASCII (Unicode) Code Points
a – z	97 – 122
A – Z	65 – 90
0 – 9	48 – 57
_	95

Although not the most expressive character set, ASCII is the most portable and least prone to corruption through improper character encodings or decodings.

Comments Allow ASCII-Compatible Encoding

Within comments, Stan can work with any ASCII-compatible character encoding, such as ASCII itself, UTF-8, or Latin1. It is up to user shells and editors to display them properly.

22.3. Parentheses for Grouping

Any expression wrapped in parentheses is also an expression. Like in C++, but unlike in R, only the round parentheses, (and), are allowed. The square brackets [and] are reserved for array indexing and the curly braces { and } for grouping statements.

With parentheses it is possible to explicitly group subexpressions with operators. Without parentheses, the expression 1 + 2 * 3 has a subexpression 2 * 3 and evaluates to 7. With parentheses, this grouping may be made explicit with the expression 1 + (2 * 3). More importantly, the expression (1 + 2) * 3 has 1 + 2 as a subexpression and evaluates to 9.

22.4. Arithmetic and Matrix Expressions

For integer and real-valued expressions, Stan supports the basic binary arithmetic operations of addition (+), subtraction (–), multiplication (∗) and division (/) in the usual ways. Stan also supports the unary operation of negation for integer and real-valued expressions. For example, assuming n and m are integer variables and x and y real variables, the following expressions are legal.

```
3.0 + 0.14, -15, 2 * 3 + 1, (x - y) / 2.0,
(n * (n + 1)) / 2, x / n
```

The negation, addition, subtraction, and multiplication operations are extended to matrices, vectors, and row vectors. The transpose operation, written using an apostrophe (') is also supported for vectors, row vectors, and matrices. Return types for matrix operations are the smallest types that can be statically guaranteed to contain the result. The full set of allowable input types and corresponding return types is detailed in Chapter 31.

For example, if y and mu are variables of type vector and Sigma is a variable of type matrix, then

```
(y - mu)' * Sigma * (y - mu)
```

is a well-formed expression of type `real`. The type of the complete expression is inferred working outward from the subexpressions. The subexpression(s) `y - mu` are of type `vector` because the variables `y` and `mu` are of type `vector`. The transpose of this expression, the subexpression `(y - mu)'` is of type `row_vector`. Multiplication is left associative and transpose has higher precedence than multiplication, so the above expression is equivalent to the following well-formed, fully specified form.

```
(((y - mu)') * Sigma) * (y - mu)
```

The type of subexpression `(y - mu)' * Sigma` is inferred to be `row_vector`, being the result of multiplying a row vector by a matrix. The whole expression's type is thus the type of a row vector multiplied by a (column) vector, which produces a `real` value.

Operator Precedence and Associativity

The precedence and associativity of operators, as well as built-in syntax such as array indexing and function application is given in tabular form in Figure 22.1. Other expression-forming operations, such as function application and subscripting bind more tightly than any of the arithmetic operations.

The precedence and associativity determine how expressions are interpreted. Because addition is left associative, the expression `a+b+c` is interpreted as `(a+b)+c`. Similarly, `a/b*c` is interpreted as `(a/b)*c`.

Because multiplication has higher precedence than addition, the expression `a*b+c` is interpreted as `(a*b)+c` and the expression `a+b*c` is interpreted as `a+(b*c)`. Similarly, `2*x+3*-y` is interpreted as `(2*x)+(3*(-y))`.

Transposition binds tighter than all other operations, so that `-u'` is interpreted as `-(u')`, `u*v'` as `u*(v')`, and `u'*v` as `(u')*v`.

22.5. Subscripting

Stan arrays, matrices, vectors, and row vectors are all accessed using the same array-like notation. For instance, if `x` is a variable of type `real[]` (a one-dimensional array of reals) then `x[1]` is the value of the first element of the array.

Subscripting has higher precedence than any of the arithmetic operations. For example, `alpha*x[1]` is equivalent to `alpha*(x[1])`.

Multiple subscripts may be provided within a single pair of square brackets. If `x` is of type `real[ , ]`, a two-dimensional array, then `x[2,501]` is of type `real`.

Op.	Prec.	Assoc.	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 22.1: *Stan's unary and binary operators, with their precedences, associativities, place in an expression, and a description. The last two lines list the precedence of function application and array, matrix, and vector indexing. The operators are listed in order of precedence, from least tightly binding to most tightly binding. The full set of legal arguments and corresponding result types are provided in the function documentation in Part V prefaced with* operator *(i.e.,* operator*(int,int):int *indicates the application of the multiplication operator to two integers, which returns an integer). Parentheses may be used to group expressions explicitly rather than relying on precedence and associativity.*

Accessing Subarrays

The subscripting operator also returns subarrays of arrays. For example, if `x` is of type `real[ , , ]`, then `x[2]` is of type `real[ , ]`, and `x[2,3]` is of type `real[]`. As a result, the expressions `x[2,3]` and `x[2][3]` have the same meaning.

Accessing Matrix Rows

If `Sigma` is a variable of type `matrix`, then `Sigma[1]` denotes the first row of `Sigma` and has the type `row_vector`.

Mixing Array and Vector/Matrix Indexes

Stan supports mixed indexing of arrays and their vector, row vector or matrix values. For example, if `m` is of type `matrix[,]`, a two-dimensional array of matrices, then `m[1]` refers to the first row of the array, which is a one-dimensional array of matrices. More than one index may be used, so that `m[1,2]` is of type `matrix` and denotes the matrix in the first row and second column of the array. Continuing to add indices, `m[1,2,3]` is of type `row_vector` and denotes the third row of the matrix denoted by `m[1,2]`. Finally, `m[1,2,3,4]` is of type `real` and denotes the value in the third row and fourth column of the matrix that is found at the first row and second column of the array `m`.

22.6. Function Application

Stan provides a broad-range of built in mathematical and statistical functions, which are documented in Part V.

Expressions in Stan may consist of the name of function followed by a sequence of zero or more argument expressions. For instance, `log(2.0)` is the expression of type `real` denoting the result of applying the natural logarithm to the value of the real literal `2.0`.

Syntactically, function application has higher precedence than any of the other operators, so that `y + log(x)` is interpreted as `y + (log(x))`.

Type Signatures and Result Type Inference

Each function has a type signature which determines the allowable type of its arguments and its return type. For instance, the function signature for the logarithm function can be expressed as

```
real log(real);
```

and the signature for the `multiply_log` function is

```
real multiply_log(real,real);
```

A function is uniquely determined by its name and its sequence of argument types. For instance, the following two functions are different functions.

```
real mean(real[]);
real mean(vector);
```

The first applies to a one-dimensional array of real values and the second to a vector.

The identity conditions for functions explicitly forbids having two functions with the same name and argument types but different return types. This restriction also makes it possible to infer the type of a function expression compositionally by only examining the type of its subexpressions.

Constants

Constants in Stan are nothing more than nullary (no-argument) functions. For instance, the mathematical constants π and e are represented as nullary functions named `pi()` and `e()`. See Section 29.1 for a list of built-in constants.

Type Promotion and Function Resolution

Because of integer to real type promotion, rules must be established for which function is called given a sequence of argument types. The scheme employed by Stan is the same as that used by C++, which resolves a function call to the function requiring the minimum number of type promotions.

For example, consider a situation in which the following two function signatures have been registered for `foo`.

```
real foo(real,real);
int foo(int,int);
```

The use of `foo` in the expression `foo(1.0,1.0)` resolves to `foo(real,real)`, and thus the expression `foo(1.0,1.0)` itself is assigned a type of `real`.

Because integers may be promoted to real values, the expression `foo(1,1)` could potentially match either `foo(real,real)` or `foo(int,int)`. The former requires two type promotions and the latter requires none, so `foo(1,1)` is resolved to function `foo(int,int)` and is thus assigned the type `int`.

The expression `foo(1,1.0)` has argument types `(int,real)` and thus does not explicitly match either function signature. By promoting the integer expression `1` to type `real`, it is able to match `foo(real,real)`, and hence the type of the function expression `foo(1,1.0)` is `real`.

In some cases (though not for any built-in Stan functions), a situation may arise in which the function referred to by an expression remains ambiguous. For example, consider a situation in which there are exactly two functions named `bar` with the following signatures.

```
real bar(real,int);
real bar(int,real);
```

With these signatures, the expression `bar(1.0,1)` and `bar(1,1.0)` resolve to the first and second of the above functions, respectively. The expression `bar(1.0,1.0)` is illegal because real values may not be demoted to integers. The expression `bar(1,1)` is illegal for a different reason. If the first argument is promoted to a real value, it matches the first signature, whereas if the second argument is promoted to a real value, it matches the second signature. The problem is that these both require one promotion, so the function name `bar` is ambiguous. If there is not a unique function requiring fewer promotions than all others, as with `bar(1,1)` given the two declarations above, the Stan compiler will flag the expression as illegal.

Random-Number Generating Functions

For most of the distributions supported by Stan, there is a corresponding random-number generating function. These random number generators are named by the distribution with the suffix `_rng`. For example, a univariate normal random number can be generated by `normal_rng(0,1)`; only the parameters of the distribution, here a location (0) and scale (1) are specified because the variate is generated.

Random-Number Generators Restricted to Generated Quantities Block

The use of random-number generating functions is restricted to the generated quantities block; attempts to use them elsewhere will result in a parsing error with a diagnostic message.

This allows the random number generating functions to be used for simulation in general, and for Bayesian posterior predictive checking in particular.

Posterior Predictive Checking

Posterior predictive checks typically use the parameters of the model to generate simulated data (at the individual and optionally at the group level for hierarchical models), which can then be compared informally using plots and formally by means of test statistics, to the actual data in order to assess the suitability of the model; see (Gelman et al., 2003, Chapter 6) for more information on posterior predictive checks.

Type	Primitive Type
int	int
real	real
matrix	matrix
cov_matrix	matrix
corr_matrix	matrix
cholesky_factor_cov	matrix

Type	Primitive Type
vector	vector
simplex	vector
unit_vector	vector
ordered	vector
positive_ordered	vector
row_vector	row_vector

Figure 22.2: *The table shows the variable declaration types of Stan and their corresponding primitive implementation type. Stan functions, operators and probability functions have argument and result types declared in terms of primitive types.*

22.7. Type Inference

Stan is strongly statically typed, meaning that the implementation type of an expression can be resolved at compile time.

Implementation Types

The primitive implementation types for Stan are `int`, `real`, `vector`, `row_vector`, and `matrix`. Every basic declared type corresponds to a primitive type; see Figure 22.2 for the mapping from types to their primitive types. A full implementation type consists of a primitive implementation type and an integer array dimensionality greater than or equal to zero. These will be written to emphasize their array-like nature. For example, `int[]` has an array dimensionality of 1, `int` an array dimensionality of 0, and `int[„]` an array dimensionality of 3. The implementation type `matrix[„]` has a total of five dimensions and takes up to five indices, three from the array and two from the matrix.

Recall that the array dimensions come before the matrix or vector dimensions in an expression such as the following declaration of a three-dimensional array of matrices.

```
    matrix[M,N] a[I,J,K];
```

The matrix `a` is indexed as `a[i,j,k,m,n]` with the array indices first, followed by the matrix indices, with `a[i,j,k]` being a matrix and `a[i,j,k,m]` being a row vector.

Type Inference Rules

Stan's type inference rules define the implementation type of an expression based on a background set of variable declarations. The rules work bottom up from primitive literal and variable expressions to complex expressions.

Literals

An integer literal expression such as `42` is of type `int`. Real literals such as `42.0` are of type `real`.

Variables

The type of a variable declared locally or in a previous block is determined by its declaration. The type of a loop variable is `int`.

There is always a unique declaration for each variable because Stan prohibits the redeclaration of an already-declared variables.[1]

Indexing

If `x` is an expression of total dimensionality greater than or equal to N, then the type of expression `e[i1,...,iN]` is the same as that of `e[i1]...[iN]`, so it suffices to define the type of a singly-indexed function. Suppose `e` is an expression and `i` is an expression of primitive type `int`. Then

- if `e` is an expression of array dimensionality $K > 0$, then `e[i]` has array dimensionality $K - 1$ and the same primitive implementation type as `e`,

- if `e` has implementation type `vector` or `row_vector` of array dimensionality 0, then `e[i]` has implementation type `real`, and

- if `e` has implementation type `matrix`, then `e[i]` has type `row_vector`.

---

[1]Languages such as C++ and R allow the declaration of a variable of a given name in a narrower scope to hide (take precedence over for evaluation) a variable defined in a containing scope. Stan will have to introduce this behavior eventually for user-defined functions written in Stan.

Function Application

If `f` is the name of a function and `e1,...,eN` are expressions for $N \geq 0$, then `f(e1,...,eN)` is an expression whose type is determined by the return type in the function signature for `f` given `e1` through `eN`. Recall that a function signature is a declaration of the argument types and the result type.

In looking up functions, binary operators like `real * real` are defined as `operator*(real,real)` in the documentation and index.

In matching a function definition, arguments of type `int` may be promoted to type `real` if necessary (see the subsection on type promotion in Section 22.6 for an exact specification of Stan's integer-to-real type-promotion rule).

In general, matrix operations return the lowest inferrable type. For example, `row_vector * vector` returns a value of type `real`, which is declared in the function documentation and index as `real operator*(row_vector,vector)`.

22.8. Chain Rule and Derivatives

Derivatives of the log probability function defined by a model are used in several ways by Stan. The Hamiltonian Monte Carlo samplers, including NUTS, use gradients to guide updates. The BFGS optimizers also use gradients to guide search for posterior modes.

Errors Due to Chain Rule

Unlike evaluations in pure mathematics, evaluation of derivatives in Stan is done by applying the chain rule on an expression-by-expression basis, evaluating using floating-point arithmetic. As a result, models such as the following are problematic for inference involving derivatives.

```
parameters {
  real x;
}
model {
  x ~ normal(sqrt(x - x), 1);
}
```

Algebraically, the sampling statement in the model could be reduced to

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

and it would seem the model should produce unit normal samples for `x`. But rather than cancelling, the expression `sqrt(x - x)` causes a problem for derivatives. The

cause is the mechanistic evaluation of the chain rule,

$$
\begin{aligned}
\frac{d}{dx}\sqrt{x-x} &= \frac{1}{2\sqrt{x-x}} \times \frac{d}{dx}(x-x) \\
&= \frac{1}{0} \times (1-1) \\
&= \infty \times 0 \\
&= \text{NaN}.
\end{aligned}
$$

Rather than the $x-x$ cancelling out, it introduces a 0 into the numerator and denominator of the chain-rule evaluation.

The only way to avoid this kind problem is to be careful to do the necessary algebraic reductions as part of the model and not introduce expressions like `sqrt(x - x)` for which the chain rule produces not-a-number values.

Diagnosing Problems with Derivatives

The best way to diagnose whether something is going wrong with the derivatives is to use the test-gradient option to the sampler or optimizer inputs; this option is available in both Stan and RStan (though it may be slow, because it relies on finite differences to make a comparison to the built-in automatic differentiation).

For example, compiling the above model to an executable `sqrt-x-minus-x`, the test can be run as

```
> ./sqrt-x-minus-x diagnose test=gradient

...
TEST GRADIENT MODE

 Log probability=-0.393734

 param idx            value          model     finite diff           error
        0        -0.887393            nan               0             nan
```

Even though finite differences calculates the right gradient of 0, automatic differentiation follows the chain rule and produces a not-a-number output.

23. Statements

The blocks of a Stan program (see Chapter 24) are made up of variable declarations and statements. Unlike programs in BUGS, the declarations and statements making up a Stan program are executed in the order in which they are written. Variables must be defined to have some value (as well as declared to have some type) before they are used — if they do not, the behavior is undefined.

Like BUGS, Stan has two kinds of atomic statements, assignment statements and sampling statements. Also like BUGS, statements may be grouped into sequences and into for-each loops. In addition, Stan allows local variables to be declared in blocks and also allows an empty statement consisting only of a semicolon.

23.1. Assignment Statement

An assignment statement consists of a variable (possibly multivariate with indexing information) and an expression. Executing an assignment statement evaluates the expression on the right-hand side and assigns it to the (indexed) variable on the left-hand side. An example of a simple assignment is

```
n <- 0;
```

Executing this statement assigns the value of the expression 0, which is the integer zero, to the variable n. For an assignment to be well formed, the type of the expression on the right-hand side should be compatible with the type of the (indexed) variable on the left-hand side. For the above example, because 0 is an expression of type `int`, the variable n must be declared as being of type `int` or of type `real`. If the variable is of type `real`, the integer zero is promoted to a floating-point zero and assigned to the variable. After the assignment statement executes, the variable n will have the value zero (either as an integer or a floating-point value, depending on its type).

Syntactically, every assignment statement must be followed by a semicolon. Otherwise, whitespace between the tokens does not matter (the tokens here being the left-hand-side (indexed) variable, the assignment operator, the right-hand-side expression and the semicolon).

Because the right-hand side is evaluated first, it is possible to increment a variable in Stan just as in C++ and other programming languages by writing

```
n <- n + 1;
```

Such self assignments are not allowed in BUGS, because they induce a cycle into the directed graphical model.

The left-hand side of an assignment may contain indices for array, matrix, or vector data structures. For instance, if `Sigma` is of type `matrix`, then

```
Sigma[1,1] <- 1.0;
```

sets the value in the first column of the first row of `Sigma` to one.

Assignments can involve complex objects of any type. If `Sigma` and `Omega` are matrices and `sigma` is a vector, then the following assignment statement, in which the expression and variable are both of type `matrix`, is well formed.

```
Sigma
  <- diag_matrix(sigma)
      * Omega
      * diag_matrix(sigma);
```

This example also illustrates the preferred form of splitting a complex assignment statement and its expression across lines.

Assignments to slices of larger multi-variate data structures are supported by Stan. For example, `a` is an array of type `real[ , ]` and `b` is an array of type `real[]`, then the following two statements are both well-formed.

```
a[3] <- b;
b <- a[4];
```

Similarly, if `x` is a variable declared to have type `row_vector` and `Y` is a variable declared as type `matrix`, then the following sequence of statements to swap the first two rows of `Y` is well formed.

```
x <- Y[1];
Y[1] <- Y[2];
Y[2] <- x;
```

In R, if `x` is a matrix or two-dimensional array, its first row is `x[1,]` and its first column is `x[,1]`. As of version 2.0, this notation is not supported by Stan. There are functions to access rows and columns of matrices, but general array slicing is not supported. Similarly, Stan 2.0 does not support providing an array of indices as an argument to create a piecemeal subarray of a larger array.

23.2. Log Probability Increment Statement

The basis of Stan's execution is the evaluation of a log probability function for a given set of parameters. Data and transformed data are fixed before log probability is

involved. Statements in the transformed parameter block and model block can have an effect on the log probability function defined by a model.

The total log probability is initialized to zero. Then statements in the transformed parameter block and model block may add to it. The most direct way this is done is through the log probability increment statement, which is of the following form.

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

In this example, the unnormalized log probability of a unit normal variable y is adeed to the total log probability. In the general case, the argument can be any expression.[1]

An entire Stan model can be implemented this way. For instance, the following model will draw a single variable according to a unit normal probability.

```
parameters {
  real y;
}
model {
  increment_log_prob(-0.5 * y * y);
}
```

This model defines a log probability function

$$\log p(y) = -\frac{y^2}{2} - \log Z$$

where Z is a normalizing constant that does not depend on y. The constant Z is conventionally written this way because on the linear scale,

$$p(y) = \frac{1}{Z} \exp\left(-\frac{y^2}{2}\right).$$

which is typically written without reference to Z as

$$p(y) \propto \exp\left(-\frac{y^2}{2}\right).$$

Stan only requires models to be defined up to a constant that does not depend on the parameters. This is convenient because often the normalizing constant Z is either time-consuming to compute or intractable to evaluate.

---

[1]Writing this model with the expression -0.5 * y * y is more efficient than with the equivalent expression y * y / -2 because multiplication is more efficient than division; in both cases, the negation is rolled into the numeric literal (-0.5 and -2). Writing square(y) instead of y * y would be even more efficient because the derivatives can be precomputed, reducing the memory and number of operations required for automatic differentiation.

Log Probability Variable `lp__`

Before version 2.0 of Stan, rather than writing

```
increment_log_prob(u);
```

it was necessary to manipulate a special variable `lp__` as follows

```
lp__ <- lp__ + u;
```

The special variable `lp__` refers to the log probability value that will be returned by Stan's log probability function.

Deprecation of `lp__`

As of Stan version 2.0, the use of `lp__` is deprecated. It will be removed altogether in a future release, but until that time, programs still work with `lp__`. A deprecation warning is now printed every time `lp__` is used.

23.3. Sampling Statements

Like BUGS and JAGS, Stan supports probability statements in sampling notation, such as

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

The name "sampling statement" is meant to be suggestive, not interpreted literally. Conceptually, the variable y, which may be an unknown parameter or known, modeled data, is being declared to have the distribution indicated by the right-hand side of the sampling statement.

Executing such a statement does not perform any sampling. In Stan, a sampling statement is merely a notational convenience. The above sampling statement could be expressed as a direct increment on the total log probability as

```
increment_log_prob(normal_log(y,mu,sigma));
```

See the subsection of Section 23.3 discussing log probability increments for a full explanation.

In general, a sampling statement of the form

```
ex0 ~ dist(ex1,...,exN);
```

involving subexpressions ex0 through exN (including the case where N is zero) will be well formed if and only if the corresponding assignment statement is well-formed,

```
        increment_log_prob(dist_log(ex0,ex1,...,exN));
```

This will be well formed if and only if `dist_log(ex0,ex1,...,exN)` is a well-formed function expression of type `real`.

Log Probability Increment vs. Sampling Statement

Although both lead to the same sampling behavior in Stan, there is one critical difference between using the sampling statement, as in

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

and explicitly incrementing the log probability function, as in

```
    increment_log_prob(normal_log(y,mu,sigma));
```

The sampling statement drops all the terms in the log probability function that are constant, whereas the explicit call to `normal_log` adds all of the terms in the definition of the log normal probability function, including all of the constant normalizing terms. Therefore, the explicit increment form can be used to recreate the exact log probability values for the model. Otherwise, the sampling statement form will be faster if any of the input expressions, `y`, `mu`, or `sigma`, involve only constants, data variables or transformed data variables.

User-Transformed Variables

The left-hand side of a sampling statement may be a complex expression. For instance, it is legal syntactically to write

```
    data {
      real<lower=0> y;
    }
    ...
    model {
      log(y) ~ normal(mu,sigma);
    }
```

Unfortunately, this is not enough to properly model `y` as having a lognormal distribution. The log Jacobian of the transform must be added to the log probability accumulator to account for the differential change in scale (see Section 49.1 for full definitions). For the case above, the following adjustment will account for the log transform.[2]

```
    increment_log_prob(- log(fabs(y)));
```

---

[2]Because $\log |\frac{d}{dy} \log y| = \log |1/y| = -\log |y|$; see Section 49.1.

Truncated Distributions

A density function $p(x)$ may be truncated to an interval (a, b) to define a new density $p_{(a,b)}(x)$ by setting

$$p_{(a,b)}(x) = \frac{p(x)}{\int_a^b p(x')\,dx'}.$$

Given a probability function $p_X(x)$ for a random variable X, its cumulative distribution function (cdf) $F_X(x)$ is defined to be the probability that $X \leq a$. For continuous random variables, this is

$$F_X(x) = \int_{-\infty}^x p_X(x')\,dx',$$

whereas for discrete variables, it's

$$F_X(x) = \sum_{n \leq x} p_X(x).$$

The complementary cumulative distribution function (ccdf) is $1 - F_X(x)$.

Cumulative distribution functions are useful for defining truncated distributions, because

$$\int_a^b p(x')\,dx' = F(b) - F(a),$$

so that

$$p_{(a,b)}(x) = \frac{p_X(x)}{F(b) - F(a)}.$$

On the log scale,

$$\log p_{(a,b)}(x) = \log p_X(x) - \log\left(F(b) - F(a)\right).$$

The denominator is more stably computed using Stan's log_diff_exp operation as

$$\log\left(F(b) - F(a)\right) = \log\left(\exp(\log F(b)) - \exp(\log F(a))\right)$$

$$= \text{log\_diff\_exp}(\log F(b), \log F(a)).$$

As in BUGS and JAGS, Stan allows probability functions to be truncated. For example, a truncated unit normal distribution restricted to $(-0.5, 2.1)$ is encoded as follows.

```
y ~ normal(0,1) T[-0.5, 2.1];
```

Truncated distributions are translated as an addition summation for the accumulated log probability. For instance, this example has the same translation (up to arithmetic precision issues; see below) as

```
increment_log_prob(normal_log(y,0,1));
increment_log_prob(-(log(normal_cdf(2.1,0,1)
                      - normal_cdf(-0.5,0,1)))));
```

The function `normal_cdf` represents the cumulative normal distribution function.
For example, `normal_cdf(2.1,0,1)` evaluates to

$$\int_{-\infty}^{2.1} \mathsf{Normal}(x|0,1)\, dx,$$

which is the probability a unit normal variable takes on values less than 2.1, or about
0.98.

Arithmetic precision is handled by working on the log scale and using Stan's log
scale cdf implementations. The logarithm of the cdf for the normal distribution is
`normal_cdf_log` and the ccdf is `normal_ccdf_log`. Stan's translation for the de-
nominator introduced by truncation is equivalent to

```
increment_log_prob(-log_diff_exp(normal_cdf_log(2.1,0,1),
                                 normal_cdf_log(-0.5,0,1)));
```

As with constrained variable declarations, truncation can be one sided. The den-
sity $p(x)$ can be truncated below by a to define a density $p_{(a,)}(x)$ with support (a, ∞)
by setting

$$p_{(a,)}(x) = \frac{p(x)}{\int_a^\infty p(x')\, dx'}.$$

For example, the unit normal distribution truncated below at -0.5 would be repre-
sented as

```
y ~ normal(0,1) T[-0.5,];
```

The truncation has the same effect as the following direct update to the accumulated
log probability (see the subsection of Section 23.3 contrasting log probability incre-
ment and sampling statements for more information).

```
increment_log_prob(normal_log(y, 0, 1));
increment_log_prob(-(1 - log(normal_cdf(-0.5, 0, 1)))));
```

The denominator is actually implemented with the more efficient and stable version

```
increment_log_prob(-normal_ccdf_log(-0.5, 0, 1));
```

The density $p(x)$ can be truncated above by b to define a density $p_{(,b)}(x)$ with
support $(-\infty, b)$ by setting

$$p_{(,b)}(x) = \frac{p(x)}{\int_{-\infty}^b p(x')\, dx'}.$$

217

For example, the unit normal distribution truncated above at 2.1 would be represented as

```
y ~ normal(0,1) T[,2.1];
```

The truncation has the same effect as the following direct update to the accumulated log probability.

```
increment_log_prob(normal_log(y, 0, 1));
increment_log_prob(-normal_cdf_log(2.1, 0, 1));
```

In all cases, the truncation is only well formed if the appropriate log cumulative distribution functions are defined.[3] Part VI and Part VII document the available discrete and continuous cumulative distribution functions. Most distributions have cdf and ccdf functions.

For continuous distributions, truncation points must be expressions of type `int` or `real`. For discrete distributions, truncation points must be expressions of type `int`.

For a truncated sampling statement, if the value sampled is not within the bounds specified by the truncation expression, the result is zero probability and the entire statement adds $-\infty$ to the total log probability, which in turn results in the sample being rejected; see the subsection of Section 10.2 discussing constraints and out-of-bounds returns for programming strategies to keep all values within bounds.

Vectorizing Truncated Distributions

Stan does not (yet) support vectorization of distribution functions with truncation.

23.4. For Loops

Suppose N is a variable of type `int`, y is a one-dimensional array of type `real[]`, and `mu` and `sigma` are variables of type `real`. Furthermore, suppose that `n` has not been defined as a variable. Then the following is a well-formed for-loop statement.

```
for (n in 1:N) {
  y[n] ~ normal(mu,sigma);
}
```

The loop variable is `n`, the loop bounds are the values in the range `1:N`, and the body is the statement following the loop bounds.

---

[3]Although most distributions have cdfs and ccdfs implemented, some cumulative distribution functions and their gradients present computational challenges because they lack simple, analytic forms.

Loop Variable Typing and Scope

The bounds in a for loop must be integers. Unlike in R, the loop is always interpreted as an upward counting loop. The range L:H will cause the loop to execute the loop with the loop variable taking on all integer values greater than or equal to L and less than or equal to H. For example, the loop for (n in 2:5) will cause the body of the for loop to be executed with n equal to 2, 3, 4, and 5, in order. The variable and bound for (n in 5:2) will not execute anything because there are no integers greater than or equal to 5 and less than or equal to 2.

Order Sensitivity and Repeated Variables

Unlike in BUGS, Stan allows variables to be reassigned. For example, the variable theta in the following program is reassigned in each iteration of the loop.

```
for (n in 1:N) {
  theta <- inv_logit(alpha + x[n] * beta);
  y[n] ~ bernoulli(theta);
}
```

Such reassignment is not permitted in BUGS. In BUGS, for loops are declarative, defining plates in directed graphical model notation, which can be thought of as repeated substructures in the graphical model. Therefore, it is illegal in BUGS or JAGS to have a for loop that repeatedly reassigns a value to a variable.[4]

In Stan, assignments are executed in the order they are encountered. As a consequence, the following Stan program has a very different interpretation than the previous one.

```
for (n in 1:N) {
  y[n] ~ bernoulli(theta);
  theta <- inv_logit(alpha + x[n] * beta);
}
```

In this program, theta is assigned after it is used in the probability statement. This presupposes it was defined before the first loop iteration (otherwise behavior is undefined), and then each loop uses the assignment from the previous iteration.

Stan loops may be used to accumulate values. Thus it is possible to sum the values of an array directly using code such as the following.

---

[4]A programming idiom in BUGS code simulates a local variable by replacing theta in the above example with theta[n], effectively creating N different variables, theta[1], ..., theta[N]. Of course, this is not a hack if the value of theta[n] is required for all n.

```
  total <- 0.0;
  for (n in 1:N)
    total <- total + x[n];
```

After the for loop is executed, the variable `total` will hold the sum of the elements in the array `x`. This example was purely pedagogical; it is easier and more efficient to write

```
  total <- sum(x);
```

A variable inside (or outside) a loop may even be reassigned multiple times, as in the following legal code.

```
  for (n in 1:100) {
    y <- y + y * epsilon;
    epsilon <- epsilon / 2.0;
    y <- y + y * epsilon;
  }
```

23.5. Conditional Statements

Stan supports full conditional statements using the same if-then-else syntax as C++. The general format is

```
  if (condition1)
    statement1
  else if (condition2)
    statement2
  ...
  else if (conditionN-1)
    statementN-1
  else
    statementN
```

There must be a single leading `if` clause, which may be followed by any number of `else if` clauses, all of which may be optionally followed by an `else` clause. Each condition must be a real or integer value, with non-zero values interpreted as true and the zero value as false.

The entire sequence of if-then-else clauses forms a single conditional statement for evaluation. The conditions are evaluated in order until one of the conditions evaluates to a non-zero value, at which point its corresponding statement is executed and the conditional statement finishes execution. If none of the conditions evaluates to a non-zero value and there is a final else clause, its statement is executed.

23.6. While Statements

Stan supports standard while loops using the same syntax as C++. The general format is as follows.

```
while (condition)
  body
```

The condition must be an integer or real expression and the body can be any statement (or sequence of statements in curly braces).

Evaluation of a while loop starts by evaluating the condition. If the condition evaluates to a false (zero) value, the execution of the loop terminates and control moves to the position after the loop. If the loop's condition evaluates to a true (non-zero) value, the body statement is executed, then the whole loop is executed again. Thus the loop is continually executed as long as the condition evaluates to a true value.

23.7. Statement Blocks and Local Variable Declarations

Just as parentheses may be used to group expressions, curly brackets may be used to group a sequence of zero or more statements into a statement block. At the beginning of each block, local variables may be declared that are scoped over the rest of the statements in the block.

Blocks in For Loops

Blocks are often used to group a sequence of statements together to be used in the body of a for loop. Because the body of a for loop can be any statement, for loops with bodies consisting of a single statement can be written as follows.

```
for (n in 1:N)
  y[n] ~ normal(mu,sigma);
```

To put multiple statements inside the body of a for loop, a block is used, as in the following example.

```
for (n in 1:N) {
  lambda[n] ~ gamma(alpha,beta);
  y[n] ~ poisson(lambda[n]);
}
```

The open curly bracket ({) is the first character of the block and the close curly bracket (}) is the last character.

Because whitespace is ignored in Stan, the following program will not compile.

```
for (n in 1:N)
  y[n] ~ normal(mu,sigma);
  z[n] ~ normal(mu,sigma); // ERROR!
```

The problem is that the body of the for loop is taken to be the statement directly following it, which is `y[n] ~ normal(mu,sigma)`. This leaves the probability statement for `z[n]` hanging, as is clear from the following equivalent program.

```
for (n in 1:N) {
  y[n] ~ normal(mu,sigma);
}
z[n] ~ normal(mu,sigma); // ERROR!
```

Neither of these programs will compile. If the loop variable `n` was defined before the for loop, the for-loop declaration will raise an error. If the loop variable `n` was not defined before the for loop, then the use of the expression `z[n]` will raise an error.

Local Variable Declarations

A for loop has a statement as a body. It is often convenient in writing programs to be able to define a local variable that will be used temporarily and then forgotten. For instance, the for loop example of repeated assignment should use a local variable for maximum clarity and efficiency, as in the following example.

```
for (n in 1:N) {
  real theta;
  theta <- inv_logit(alpha + x[n] * beta);
  y[n] ~ bernoulli(theta);
}
```

The local variable `theta` is declared here inside the for loop. The scope of a local variable is just the block in which it is defined. Thus `theta` is available for use inside the for loop, but not outside of it. As in other situations, Stan does not allow variable hiding. So it is illegal to declare a local variable `theta` if the variable theta is already defined in the scope of the for loop. For instance, the following is not legal.

```
for (m in 1:M) {
  real theta;
  for (n in 1:N) {
```

```
      real theta; // ERROR!
      theta <- inv_logit(alpha + x[m,n] * beta);
      y[m,n] ~ bernoulli(theta);
   ...
```

The compiler will flag the second declaration of `theta` with a message that it is already defined.

No Constraints on Local Variables

Local variables may not have constraints on their declaration. The only types that may be used are

> `int`, `real`, `vector[K]`, `row_vector[K]`, and `matrix[M,N]`.

Blocks within Blocks

A block is itself a statement, so anywhere a sequence of statements is allowed, one or more of the statements may be a block. For instance, in a for loop, it is legal to have the following

```
for (m in 1:M) {
  {
    int n;
    n <- 2 * m;
    sum <- sum + n
  }
  for (n in 1:N)
    sum <- sum + x[m,n];
}
```

The variable declaration `int n;` is the first element of an embedded block and so has scope within that block. The for loop defines its own local block implicitly over the statement following it in which the loop variable is defined. As far as Stan is concerned, these two uses of `n` are unrelated.

23.8. Print Statements

Stan provides print statements that can print literal strings and the values of expressions. Print statements accept any number of arguments. Consider the following for-each statement with a print statement in its body.

```
for (n in 1:N) { print("loop iteration: ", n); ... }
```

The print statement will execute every time the body of the loop does. Each time the loop body is executed, it will print the string "loop iteration: " (with the trailing space), followed by the value of the expression n, followed by a new line.

Print Content

The text printed by a print statement varies based on its content. A literal (i.e., quoted) string in a print statement always prints exactly that string (without the quotes). Expressions in print statements result in the value of the expression being printed. But how the value of the expression is formatted will depend on its type.

Printing a simple `real` or `int` typed variable always prints the variable's value.[5] For array, vector, and matrix variables, the print format uses brackets. For example, a 3-vector will print as

```
[1,2,3]
```

and a 2×3-matrix as

```
[[1,2,3],[4,5,6]]
```

Printing a more readable version of arrays or matrices can be done with loops. An example is the print statement in the following transformed data block.

```
transformed data {
  matrix[2,2] u;
  u[1,1] <- 1.0;  u[1,2] <- 4.0;
  u[2,1] <- 9.0;  u[2,2] <- 16.0;
  for (n in 1:2)
    print("u[", n, "] = ", u[n]);
}
```

This print statement executes twice, printing the following two lines of output.

```
u[1] = [1,4]
u[2] = [9,16]
```

---

[5]The adjoint component is always zero during execution for the algorithmic differentiation variables used to implement parameters, transformed parameters, and local variables in the model.

Print Frequency

Printing for a print statement happens every time it is executed. The `transformed data` block is executed once per chain, the `transformed parameter` and `model` blocks once per leapfrog step, and the `generated quantities` block once per iteration.

String Literals

String literals begin and end with a double quote character (`"`). The characters between the double quote characters may be the space character or any visible ASCII character, with the exception of the backslash character (`\`) and double quote character (`"`). The full list of visible ASCII characters is as follows.

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
0 1 2 3 4 5 6 7 8 9 0 ~ @ # $ % ^ & * _ ' ` - + = {
} [ ] ( ) < > | / ! ? . , ; :
```

Debug by `print`

Because Stan is an imperative language, print statements can be very useful for debugging. They can be used to display the values of variables or expressions at various points in the execution of a program. They are particularly useful for spotting problematic not-a-number of infinite values, both of which will be printed.

24. Program Blocks

A Stan program is organized into a sequence of named blocks, the bodies of which consist of variable declarations, followed in the case of some blocks with statements.

24.1. Comments

Stan supports C++-style line-based and bracketed comments. Comments may be used anywhere whitespace is allowed in a Stan program.

Line-Based Comments

In line-based comments, any text on a line following two forward slashes (//) or the pound sign (#) is ignored (along with the slashes or pound sign).

Bracketed Comments

For bracketed comments, any text between a forward-slash and asterisk pair (/*) and an asterisk and forward-slash pair (*/) is ignored.

Character Encoding

Comments may be in ASCII, UTF-8, Latin1, or any other character encoding that is byte-wise compatible with ASCII. This excludes encodings like UTF-16, Big5, etc.[1]

24.2. Overview of Stan's Program Blocks

The full set of named program blocks is exemplified in the following skeletal Stan program.

```
data {
  ... declarations ...
}
transformed data {
   ... declarations ... statements ...
}
parameters {
   ... declarations ...
```

---

[1] The issue is that they will separate the characters in */ and */.

```
    }
    transformed parameters {
        ... declarations ... statements ...
    }
    model {
        ... declarations ... statements ...
    }
    generated quantities {
        ... declarations ... statements ...
    }
```

Optionality and Ordering

All of the blocks other than the `model` block are optional. The blocks that occur must occur in the order presented in the skeletal program above. Within each block, both declarations and statements are optional, subject to the restriction that the declarations come before the statements.

Variable Scope

The variables declared in each block have scope over all subsequent statements. Thus a variable declared in the transformed data block may be used in the model block. But a variable declared in the generated quantities block may not be used in any earlier block, including the model block.

Automatic Variable Definitions

The variables declared in the `data` and `parameters` block are treated differently than other variables in that they are automatically defined by the context in which they are used. This is why there are no statements allowed in the data or parameters block.

The variables in the `data` block are read from an external input source such as a file or a designated R data structure. The variables in the `parameters` block are read from the sampler's current parameter values (either standard HMC or NUTS). The initial values may be provided through an external input source, which is also typically a file or a designated R data structure. In each case, the parameters are instantiated to the values for which the model defines a log probability function.

Transformed Variables

The `transformed data` and `transformed parameters` block behave similarly to each other. Both allow new variables to be declared and then defined through a

sequence of statements. Because variables scope over every statement that follows them, transformed data variables may be defined in terms of the data variables.

Before generating any samples, data variables are read in, then the transformed data variables are declared and the associated statements executed to define them. This means the statements in the transformed data block are only ever evaluated once.[2] Transformed parameters work the same way, being defined in terms of the parameters, transformed data, and data variables. The difference is the frequency of evaluation. Parameters are read in and (inverse) transformed to constrained representations on their natural scales once per log probability and gradient evaluation. This means the inverse transforms and their log absolute Jacobian determinants are evaluated once per leapfrog step. Transformed parameters are then declared and their defining statements executed once per leapfrog step.

Generated Quantities

The generated quantity variables are defined once per sample after all the leapfrog steps have been completed. These may be random quantities, so the block must be rerun even if the Metropolis adjustment of HMC or NUTS rejects the update proposal.

Variable Read, Write, and Definition Summary

A table summarizing the point at which variables are read, written, and defined is given in Figure 24.1. Another way to look at the variables is in terms of their function. To decide which variable to use, consult the charts in Figure 24.2. The last line has no corresponding location, as there is no need to print a variable every iteration that does not depend on parameters.[3] The rest of this chapter provides full details on when and how the variables and statements in each block are executed.

24.3. Statistical Variable Taxonomy

(Gelman and Hill, 2007, p. 366) provides a taxonomy of the kinds of variables used in Bayesian models. Figure 24.3 contains Gelman and Hill's taxonomy along with a missing-data kind along with the corresponding locations of declarations and definitions in Stan.

Constants can be built into a model as literals, data variables, or as transformed data variables. If specified as variables, their definition must be included in data files.

---

[2]If the C++ code is configured for concurrent threads, the data and transformed data blocks can be executed once and reused for multiple chains.

[3]It is possible to print a variable every iteration that does not depend on parameters — just define it (or redefine it if it is transformed data) in the `generated quantities` block.

Block	Stmt	Action / Period
`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 step
`generated quantities`	yes	eval / sample
		write / sample
(initialization)	n/a	read, transform / chain

Figure 24.1: *The read, write, transform, and evaluate actions and periodicities listed in the last column correspond to the Stan program blocks in the first column. The middle column indicates whether the block allows statements. The last row indicates that parameter initialization requires a read and transform operation applied once per chain.*

If they are specified as transformed data variables, they cannot be used to specify the sizes of elements in the `data` block.

The following program illustrates various variables kinds, listing the kind of each variable next to its declaration.

```
data {
  int<lower=0> N;           // unmodeled data
  real y[N];                // modeled data
  real mu_mu;               // config. unmodeled param
  real<lower=0> sigma_mu;   // config. unmodeled param
}
transformed data {
  real<lower=0> alpha;      // const. unmodeled param
  real<lower=0> beta;       // const. unmodeled param
  alpha <- 0.1;
  beta <- 0.1;
}
parameters {
  real mu_y;                // modeled param
  real<lower=0> tau_y;      // modeled param
}
transformed parameters {
  real<lower=0> sigma_y;    // derived quantity (param)
```

Params	Log Prob	Print	Declare In
+	+	+	`transformed parameters`
+	+	−	*local in* `model`
+	−	−	*local in* `generated quantities`
+	−	+	`generated quantities`
−	−	+	`generated quantities`*
−	±	−	*local in* `transformed data`
−	+	+	`transformed data` *and* `generated quantities`*

Figure 24.2: *This table indicates where variables that are not basic data or parameters should be declared, based on whether it is defined in terms of parameters, whether it is used in the log probability function defined in the model block, and whether it is printed. The two lines marked with asterisks (∗) should not be used as there is no need to print a variable every iteration that does not depend on the value of any parameters (for information on how to print these if necessary, see Footnote 3 in this chapter).*

```
    sigma_y <- pow(tau_y,-0.5);
}
model {
  tau_y ~ gamma(alpha,beta);
  mu_y ~ normal(mu_mu,sigma_mu);
  for (n in 1:N)
    y[n] ~ normal(mu_y,sigma_y);
}
generated quantities {
  real variance_y;         // derived quantity (transform)
  variance_y <- sigma_y * sigma_y;
}
```

In this example, `y[N]` is a modeled data vector. Although it is specified in the `data` block, and thus must have a known value before the program may be run, it is modeled as if it were generated randomly as described by the model.

The variable `N` is a typical example of unmodeled data. It is used to indicate a size that is not part of the model itself.

The other variables declared in the data and transformed data block are examples of unmodeled parameters, also known as hyperparameters. Unmodeled parameters are parameters to probability densities that are not themselves modeled probabilistically. In Stan, unmodeled parameters that appear in the `data` block may be specified on a per-model execution basis as part of the data read. In the above model, `mu_mu` and `sigma_mu` are configurable unmodeled parameters.

Variable Kind	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	loop statement

Figure 24.3: *Variables of the kind indicated in the left column must be declared in one of the blocks declared in the right column.*

Unmodeled parameters that are hard coded in the model must be declared in the `transformed data` block. For example, the unmodeled parameters `alpha` and `beta` are both hard coded to the value 0.1. To allow such variables to be configurable based on data supplied to the program at run time, they must be declared in the `data` block, like the variables `mu_mu` and `sigma_mu`.

This program declares two modeled parameters, `mu` and `tau_y`. These are the location and precision used in the normal model of the values in `y`. The heart of the model will be sampling the values of these parameters from their posterior distribution.

The modeled parameter `tau_y` is transformed from a precision to a scale parameter and assigned to the variable `sigma_y` in the `transformed parameters` block. Thus the variable `sigma_y` is considered a derived quantity — its value is entirely determined by the values of other variables.

The `generated quantities` block defines a value `variance_y`, which is defined as a transform of the scale or deviation parameter `sigma_y`. It is defined in the generated quantities block because it is not used in the model. Making it a generated quantity allows it to be monitored for convergence (being a non-linear transform, it will have different autocorrelation and hence convergence properties than the deviation itself).

In later versions of Stan which have random number generators for the distributions, the `generated quantities` block will be usable to generate replicated data for model checking.

Finally, the variable `n` is used as a loop index in the `model` block.

24.4. Program Block: `data`

The rest of this chapter will lay out the details of each block in order, starting with the `data` block in this section.

Variable Reads and Transformations

The `data` block is for the declaration of variables that are read in as data. With the current model executable, each Markov chain of samples will be executed in a different process, and each such process will read the data exactly once.[4]

Data variables are not transformed in any way. The format for data files is provided in Chapter 6.

Statements

The `data` block does not allow statements.

Variable Constraint Checking

Each variable's value is validated against its declaration as it is read. For example, if a variable `sigma` is declared as `real<lower=0>`, then trying to assign it a negative value will raise an error. As a result, data type errors will be caught as early as possible. Similarly, attempts to provide data of the wrong size for a compound data structure will also raise an error.

24.5. Program Block: `transformed data`

The `transformed data` block is for declaring and defining variables that do not need to be changed when running the program.

Variable Reads and Transformations

For the `transformed data` block, variables are all declared in the variable declarations and defined in the statements. There is no reading from external sources and no transformations performed.

Variables declared in the `data` block may be used to declare transformed variables.

---

[4]With multiple threads, or even running chains sequentially in a single thread, data could be read only once per set of chains. Stan was designed to be thread safe and future versions will provide a multithreading option for Markov chains.

Statements

The statements in a `transformed data` block are used to define (provide values for) variables declared in the `transformed data` block. Assignments are only allowed to variables declared in the `transformed data` block.

These statements are executed once, in order, right after the data is read into the data variables. This means they are executed once per chain (though see Footnote 4 in this chapter).

Variables declared in the `data` block may be used in statements in the `transformed data` block.

Restriction on Operations in `transformed data`

The statements in the transformed data block are designed to be executed once and have a deterministic result. Therefore, log probability is not accumulated and sampling statements may not be used. Random number generating functions are also prohibited.

Variable Constraint Checking

Any constraints on variables declared in the `transformed data` block are checked after the statements are executed. If any defined variable violates its constraints, Stan will halt with a diagnostic error message.

24.6. Program Block: `parameters`

The variables declared in the `parameters` program block correspond directly to the variables being sampled by Stan's samplers (HMC and NUTS). From a user's perspective, the parameters in the program block *are* the parameters being sampled by Stan.

Variables declared as parameters cannot be directly assigned values. So there is no block of statements in the `parameters` program block. Variable quantities derived from parameters may be declared in the `transformed parameters` or `generated quantities` blocks, or may be defined as local variables in any statement blocks following their declaration.

There is a substantial amount of computation involved for parameter variables in a Stan program at each leapfrog step within the HMC or NUTS samplers, and a bit more computation along with writes involved for saving the parameter values corresponding to a sample.

Constraining Inverse Transform

Stan's two samplers, standard Hamiltonian Monte Carlo (HMC) and the adaptive No-U-Turn sampler (NUTS), are most easily (and often most effectively) implemented over a multivariate probability density that has support on all of $\mathbb{R}^n$. To do this, the parameters defined in the `parameters` block must be transformed so they are unconstrained.

In practice, the samplers keep an unconstrained parameter vector in memory representing the current state of the sampler. The model defined by the compiled Stan program defines an (unnormalized) log probability function over the unconstrained parameters. In order to do this, the log probability function must apply the inverse transform to the unconstrained parameters to calculate the constrained parameters defined in Stan's `parameters` program block. The log Jacobian of the inverse transform is then added to the accumulated log probability function. This then allows the Stan model to be defined in terms of the constrained parameters.

In some cases, the number of parameters is reduced in the unconstrained space. For instance, a K-simplex only requires $K - 1$ unconstrained parameters, and a K-correlation matrix only requires $\binom{K}{2}$ unconstrained parameters. This means that the probability function defined by the compiled Stan program may have fewer parameters than it would appear from looking at the declarations in the `parameters` program block.

The probability function on the unconstrained parameters is defined in such a way that the order of the parameters in the vector corresponds to the order of the variables defined in the `parameters` program block. The details of the specific transformations are provided in Chapter 49.

Gradient Calculation

Hamiltonian Monte Carlo requires the gradient of the (unnormalized) log probability function with respect to the unconstrained parameters to be evaluated during every leapfrog step. There may be one leapfrog step per sample or hundreds, with more being required for models with complex posterior distribution geometries.

Gradients are calculated behind the scenes using Stan's algorithmic differentiation library. The time to compute the gradient does not depend directly on the number of parameters, only on the number of subexpressions in the calculation of the log probability. This includes the expressions added from the transforms' Jacobians.

The amount of work done by the sampler does depend on the number of unconstrained parameters, but this is usually dwarfed by the gradient calculations.

Writing Samples

In the basic Stan compiled program, the values of variables are written to a file for each sample. The constrained versions of the variables are written, again in the order they are defined in the `parameters` block. In order to do this, the transformed parameter, model, and generated quantities statements must be executed.

24.7. Program Block: `transformed parameters`

The `transformed parameters` program block consists of optional variable declarations followed by statements. After the statements are executed, the constraints on the transformed parameters are validated. Any variable declared as a transformed parameter is part of the output produced for samples.

Any variable that is defined wholly in terms of data or transformed data should be declared and defined in the transformed data block. Defining such quantities in the transformed parameters block is legal, but much less efficient than defining them as transformed data.

24.8. Program Block: `model`

The `model` program block consists of optional variable declarations followed by statements. The variables in the model block are local variables and are not written as part of the output.

Local variables may not be defined with constraints because there is no well-defined way to have them be both flexible and easy to validate.

The statements in the model block typically define the model. This is the block in which probability (sampling notation) statements are allowed. These are typically used when programming in the BUGS idiom to define the probability model.

24.9. Program Block: `generated quantities`

The `generated quantities` program block is rather different than the other blocks. Nothing in the generated quantities block affects the sampled parameter values. The block is executed only after a sample has been generated.

Among the applications of posterior inference that can be coded in the generated quantities block are

- forward sampling to generate simulated data for model testing,

- generating predictions for new data,

- calculating posterior event probabilities, including multiple comparisons, sign tests, etc.,

- calculating posterior expectations,

- transforming parameters for reporting,

- applying full Bayesian decision theory,

- calculating log likelihoods, deviances, etc. for model comparison.

Forward samples, event probabilities and statistics may all be calculated directly using plug-in estimates. Stan automatically provides full Bayesian inference by producing samples from the posterior distribution of any calculated event probabilities, predictions, or statistics. See Chapter 47 for more information on Bayesian inference.

Within the generated quantities block, the values of all other variables declared in earlier program blocks (other than local variables) are available for use in the generated quantities block.

It is more efficient to define a variable in the generated quantities block instead of the transformed parameters block. Therefore, if a quantity does not play a role in the model, it should be defined in the generated quantities block.

After the generated quantities statements are executed, the constraints on the declared generated quantity variables are validated.

All variables declared as generated quantities are printed as part of the output.

25.   Modeling Language Syntax

This chapter defines the basic syntax of the Stan modeling language using a Backus-Naur form (BNF) grammar plus extra-grammatical constraints on function typing and operator precedence and associativity.

25.1.   BNF Grammars

Syntactic Conventions

In the following BNF grammars, literal strings are indicated in single quotes ('). Grammar non-terminals are unquoted strings. A prefix question mark (?) indicates optionality. A postfixed Kleene star (∗) indicates zero or more occurrences.

Programs

```
program ::= ?data ?tdata ?params ?tparams model ?generated

data ::= 'data' var_decls
tdata ::= 'transformed data' var_decls_statements
params ::= 'parameters' var_decls
tparams ::= 'transformed parameters' var_decls_statements
model ::= 'model' statement
generated ::= 'generated quantities' var_decls_statements

var_decls ::= '{' var_decl* '}'
var_decls_statements ::= '{' var_decl* statement* '}'
```

Variable Declarations

```
var_decl ::= var_type variable ?dims

var_type ::= 'int' range_constraint
           | 'real' range_constraint
           | 'vector' range_constraint '[' expression ']'
           | 'ordered' '[' expression ']'
           | 'positive_ordered' '[' expression ']'
           | 'simplex' '[' expression ']'
           | 'unit_vector' '[' expression ']'
           | 'row_vector' range_constraint '[' expression ']'
           | 'matrix' range_constraint '[' expression ',' expression ']'
           | 'cholesky_factor_cov' '[' expression ?(',' expression) ']'
```

```
             | 'corr_matrix' '[' expression ']'
             | 'cov_matrix' '[' expression ']'

range_constraint ::= ?('<' range '>')

range ::= 'lower' '=' expression ',' 'upper' = expression
        | 'lower' '=' expression
        | 'upper' '=' expression

dims ::= '['  expression (',' expression)*  ']'

variable ::= identifier

identifier ::= [a-zA-Z] [a-zA-Z0-9_]*
```

Expressions

```
expression ::= numeric_literal
             | variable
             | expression infixOp expression
             | prefixOp expression
             | expression postfixOp
             | expression '[' expressions ']'
             | function_literal '(' ?expressions ')'
             | '(' expression ')'

expressions ::= expression
              | expression ',' expressions

numeric_literal ::= int_literal | real_literal

integer_literal ::= [0-9]*

real_literal ::= [0-9]* ?('.' [0-9]*) ?exp_literal

exp_literal ::= ('e' | 'E') integer_literal

function_literal ::= identifier
```

Statements

```
statement
::= lhs '<-' expression ';'
  | expression '~' identifier '(' ?expressions ')' ?truncation ';'
```

```
  | 'if' '(' expression ')' statement
    ('else' 'if' '(' expression ')' statement)*
    ?('else' statement)
  | 'while' '(' expression ')' statement
  | 'for' '(' identifier 'in' expression ':' expression ')' statement
  | '{' var_decl* statement+ '}'
  | 'print' '(' (expression | string_literal)* ')'
  | ';'

string_literal ::= '"' char* '"'

truncation ::= 'T' '[' ?expression ',' ?expression ']'

lhs ::= identifier
      | identifier '[' expressions ']'
```

25.2. Extra-Grammatical Constraints

Type Constraints

A well-formed Stan program must satisfy the type constraints imposed by functions and distributions. For example, the binomial distribution requires an integer total count parameter and integer variate and when truncated would require integer truncation points. If these constraints are violated, the program will be rejected during parsing with an error message indicating the location of the problem. For information on argument types, see Part V.

Operator Precedence and Associativity

In the Stan grammar provided in this chapter, the expression 1 + 2 * 3 has two parses. As described in Section 22.4, Stan disambiguates between the meaning $1 + (2 \times 3)$ and the meaning $(1 + 2) \times 3$ based on operator precedences and associativities.

Forms of Numbers

Integer literals longer than one digit may not start with 0 and real literals cannot consist of only a period or only an exponent.

Conditional Arguments

Both the conditional if-then-else statement and while-loop statement require the expression denoting the condition to be a primitive type, integer or real.

Part V

Built-In Functions

26.   Vectorization

Stan's scalar log probability functions all support vectorized function application, with results defined to be the sum of the elementwise application of the function. In all cases, matrix operations are faster than loops and vectorized log probability functions are faster than their equivalent form defined with loops.

Stan also overloads some scalar functions, such as `log` and `exp`, to apply to vectors (arrays) and return vectors (arrays). These vectorizations are defined elementwise and unlike the probability functions, provide only minimal efficiency speedups over repeated application and assignment in a loop.

26.1.   Vectorized Function Signatures

The normal probability function is specified with the signature

```
normal_log(reals,reals,reals);
```

The pseudo-type `reals` is used to indicate that an argument position may be vectorized. Argument positions declared as `reals` may be filled with a real, a one-dimensional array, a vector, or a row-vector. If there is more than one array or vector argument, their types can be anything but their size must match. For instance, it is legal to use `normal_log(row_vector,vector,real)` as long as the vector and row vector have the same size.

The pseudo-type `ints` is used for vectorized integer arguments.

26.2.   Evaluating Vectorized Functions

The result of a vectorized log probability function is equivalent to the sum of the evaluations on each element. Any non-vector argument, namely `real` or `int`, is repeated. For instance, if `y` is a vector of size N, `mu` is a vector of size N, and `sigma` is a scalar, then

```
ll <- normal_log(y, mu, sigma);
```

is just a more efficient way to write

```
ll <- 0;
for (n in 1:N)
  ll <- ll + normal_log(y[n], mu[n], sigma);
```

With the same arguments, the vectorized sampling statement

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

has the same effect on the total log probability as

```
for (n in 1:N)
  y[n] ~ normal(mu[n], sigma);
```

27. Void Functions

Stan does not technically support functions that do not return values. It does support two types of statements that look like functions, one for incrementing log probabilities and one for printing. Documentation on these functions is included here for completeness.

Although it's not part of Stan's type language, in this chapter, `void` will be used for the return type.

27.1. Increment Log Probability

There is a special function `increment_log_prob` takes a single real-valued argument and adds the result to the log probability.

void **increment_log_prob**(real *lp*)

> Add *lp* to the total log probability accumulator returned by the log probability function defined by a Stan model.

The full behavior of the `increment_log_prob` statement and its relation to sampling statements is described in Section 23.2.

27.2. Print

The `print` statement is unique among Stan's syntactic constructs in two ways. First, it is the only function-like construct that allows a variable number of arguments. Second, it is the only function-like construct to accept string literals (e.g., `"hello world"`) as arguments.

Printing has no effect on the model's log probability function. Its sole purpose is the side effect (i.e., an effect not represented in a return value) of arguments being printed to whatever the standard output stream is connected to (e.g., the terminal in command-line Stan or the R console in RStan).

void **print**(T1 *x1*,..., TN *xN*)

> Print the values denoted by the arguments *x1* through *xN* on the standard output stream. There are no spaces between items in the print, but a line feed (LF; Unicode U+000A; C++ literal '\n') is inserted at the end of the printed line. The types T1 through TN can be any of Stan's built-in numerical types or double quoted strings of ASCII characters.

The full behavior of the `print` statement with examples is documented in Section 23.8.

28. Integer-Valued Basic Functions

This chapter describes Stan's built-in function that take various types of arguments and return results of type integer.

28.1. Integer-Valued Arithmetic Operators

Stan's arithmetic is based on standard double-precision C++ integer and floating-point arithmetic. If the arguments to an arithmetic operator are both integers, as in 2 + 2, integer arithmetic is used. If one argument is an integer and the other a floating-point value, as in 2.0 + 2 and 2 + 2.0, then the integer is promoted to a floating point value and floating-point arithmetic is used.

Integer arithmetic behaves slightly differently than floating point arithmetic. The first difference is how overflow is treated. If the sum or product of two integers overflows the maximum integer representable, the result is an undesirable wraparound behavior at the bit level. If the integers were first promoted to real numbers, they would not overflow a floating-point representation. There are no extra checks in Stan to flag overflows, so it is up to the user to make sure it does not occur.

Secondly, because the set of integers is not closed under division and there is no special infinite value for integers, integer division implicitly rounds the result. If both arguments are positive, the result is rounded down. For example, 1 / 2 evaluates to 0 and 5 / 3 evaluates to 1.

If one of the integer arguments to division is negative, the latest C++ specification (C++11), requires rounding toward zero. This would have -1 / 2 evaluate to 0 and -7 / 2 evaluate to 3. Before the C++11 specification, the behavior was platform dependent, allowing rounding up or down. All compilers recent enough to be able to deal with Stan's templating should follow the C++11 specification, but it may be worth testing if you are not sure and plan to use integer division with negative values.

Unlike floating point division, where 1.0 / 0.0 produces the special positive infinite value, integer division by zero, as in 1 / 0, has undefined behavior in the C++ standard. For example, the clang++ compiler on Mac OS X returns 3764, whereas the g++ compiler throws an exception and aborts the program with a warning. As with overflow, it is up to the user to make sure integer divide-by-zero does not occur.

Binary Infix Operators

Operators are described using the C++ syntax. For instance, the binary operator of addition, written X + Y, would have the Stan signature int operator+(int,int) indicating it takes two real arguments and returns a real value.

```
int operator+(int x, int y)
```
> The sum of the addends x and y

```
int operator-(int x, int y)
```
> The difference between the minuend x and subtrahend y

```
int operator*(int x, int y)
```
> The product of the factors x and y

```
int operator/(int x, int y)
```
> The integer quotient of the dividend x and divisor y

Unary Prefix Operators

```
int operator-(int x)
```
> The negation of the subtrahend x

```
int operator+(int x)
```
> This is a no-op.

28.2.   Absolute Functions

```
int abs(int x)
```
> The absolute value of x

```
int int_step(int x)
```
> 1 if x is strictly greater than 0, and 0 otherwise

```
int int_step(real x)
```
> 1 if x is strictly greater than 0, and 0 otherwise

28.3.   Bound Functions

```
int min(int x, int y)
```
> The minimum of x and y

```
int max(int x, int y)
```
> The maximum of x and y

29. Real-Valued Basic Functions

This chapter describes built-in functions that take zero or more real or integer arguments and return real values.

29.1. Mathematical Constants

Constants are represented as functions with no arguments and must be called as such. For instance, the mathematical constant π must be written in a Stan program as `pi()`.

real **pi**()
> π, the ratio of a circle's circumference to its diameter

real **e**()
> e, the base of the natural logarithm

real **sqrt2**()
>   The square root of 2

real **log2**()
>   The natural logarithm of 2

real **log10**()
>   The natural logarithm of 10

29.2. Special Values

real **not_a_number**()
>   Not-a-number, a special non-finite real value returned to signal an error

real **positive_infinity**()
>   Positive infinity, a special non-finite real value larger than all finite numbers

real **negative_infinity**()
>   Negative infinity, a special non-finite real value smaller than all finite numbers

real **machine_precision**()
>   The smallest number x such that $(x + 1) \neq 1$ in floating-point arithmetic on the current hardware platform

29.3. Logical Functions

Like C++, BUGS, and R, Stan uses 0 to encode false, and 1 to encode true. Stan supports the usual boolean comparison operations and boolean operators. These all have the same syntax and precedence as in C++; for the full list of operators and precedences, see Figure 22.1.

Comparison Operators

All comparison operators return boolean values, either 0 or 1. Each operator has two signatures, one for integer comparisons and one for floating-point comparisons. Comparing an integer and real value is carried out by first promoting the integer value.

int **operator<**(int *x*, int *y*)
    1 if *x* is less than *y* and 0 otherwise

int **operator<=**(int *x*, int *y*)
    1 if *x* is less than or equal to *y* and 0 otherwise

int **operator>**(int *x*, int *y*)
    1 if *x* is greater than *y* and 0 otherwise

int **operator>=**(int *x*, int *y*)
    1 if *x* is greater than or equal to *y* and 0 otherwise

int **operator==**(int *x*, int *y*)
    1 if *x* is equal to *y* and 0 otherwise

int **operator!=**(int *x*, int *y*)
    1 if *x* is not equal to *y* and 0 otherwise

The real-valued argument versions are identical other than for argument type.

int **operator<**(real *x*, real *y*)
    1 if *x* is less than *y* and 0 otherwise

int **operator<=**(real *x*, real *y*)
    1 if *x* is less than or equal to *y* and 0 otherwise

int **operator>**(real *x*, real *y*)
    1 if *x* is greater than *y* and 0 otherwise

```
int operator>=(real x, real y)
```
1 if *x* is greater than or equal to *y* and 0 otherwise

```
int operator==(real x, real y)
```
1 if *x* is equal to *y* and 0 otherwise

```
int operator!=(real x, real y)
```
1 if *x* is not equal to *y* and 0 otherwise

Boolean Operators

Boolean operators return either 0 for false or 1 for true. Inputs may be any real or integer values, with non-zero values being treated as true and zero values treated as false. These operators have the usual precedences, with negation (not) binding the most tightly, conjunction the next and disjunction the weakest; all of the operators bind more tightly than the comparisons. Thus an expression such as `!a~&&~b` is interpreted as `(!a)~&&~b`, and `a~<~b~||~c~>=~d~&&~e~!=~f` as `(a~<~b)~||~(((c~>=~d)~&&~(e~!=~f)))`.

```
int operator!(int x)
```
1 if *x* is zero and 0 otherwise

```
int operator&&(int x, int y)
```
1 if *x* is unequal to 0 and *y* is unequal to 0

```
int operator||(int x, int y)
```
1 if *x* is unequal to 0 or *y* is unequal to 0

There are corresponding real-argument versions.

```
int operator!(real x)
```
1 if *x* is zero and 0 otherwise

```
int operator&&(real x, real y)
```
1 if *x* is unequal to 0 and *y* is unequal to 0

```
int operator||(real x, real y)
```
1 if *x* is unequal to 0 or *y* is unequal to 0

Boolean Operator Short Circuiting

Like in C++, the boolean operators are implemented to short circuit directly to a return value after evaluating the first argument if it is sufficient to resolve the result. In evaluating `a || b`, if `a` evaluates to a value other than zero, the expression returns the value 1 without evaluating the expression `b`. Similarly, evaluating `a && b` first evaluates `a`, and if the result is zero, returns 0 without evaluating `b`.

Logical Functions

The logical functions introduce conditional behavior functionally and are primarily provided for compatibility with BUGS and JAGS.

real **if_else**(int *cond*, real *x*, real *y*)

    x if *cond* is non-zero, and *y* otherwise; unlike the ternary operator in C++, Stan's `if_else` function always evaluates both arguments *x* and *y*

real **step**(real *x*)

    1 if *x* is positive and 0 otherwise; equivalent to `x > 0.0`

The log probability function and gradient evaluations are more efficient in Stan when implemented using conditional statements. If `y` is a `real` variable, and `c`, `x1`, and `x2` are scalar expressions (type `real` or `int`), then the assignment statements

```
y <- x1 * step(c) + x2 * (1 - step(c));
```

and

```
y <- if_else(c > 0, x1, x2);
```

are more efficiently written with the conditional statement

```
if (c > 0)
  y <- x1;
else
  y <- x2;
```

The reason the functional versions are slower is that they evaluate all of their arguments; the step function approach is particularly slow as it also introduces arithmetic operations. The overhead will be more noticeable if `c`, `x1` or `x2` are parameters (including transformed parameters and local variables that depend on parameters) or if `x1` and `x2` are complicated expressions rather than constants or simple variables.

    Warning: If `y` is a parameter (including transformed parameters and local variables in model blocks) and any of `c`, `x1`, or `x2` is a parameter, then all of the above approaches introduce the same discontinuities into the derivative of `y` with respect to the parameter arguments.

29.4. Real-Valued Arithmetic Operators

The arithmetic operators are presented using C++ notation. For instance `operator+(x,y)` refers to the binary addition operator and `operator-(x)` to the unary negation operator. In Stan programs, these are written using the usual infix and prefix notations as `x + y` and `-x`, respectively.

Binary Infix Operators

real **operator+**(real *x*, real *y*)
> The sum of the addends *x* and *y*

real **operator-**(real *x*, real *y*)
> The difference between the minuend *x* and subtrahend *y*

real **operator***(real *x*, real *y*)
> The product of the factors *x* and *y*

real **operator/**(real *x*, real *y*)
> The quotient of the dividend *x* and divisor *y*

Unary Prefix Operators

real **operator-**(real *x*)
> The negation of the subtrahend *x*

real **operator+**(real *x*)
> This is a no-op.

29.5. Step-like Functions

Warning: *These functions can seriously hinder sampling and optimization efficiency for gradient-based methods (e.g., NUTS, HMC, BFGS) if applied to parameters (including transformed parameters and local variables in the transformed parameters or model block). The problem is that they break gradients due to discontinuities coupled with zero gradients elsewhere. They do not hinder sampling when used in the data, transformed data, or generated quantities blocks.*

Absolute Value Functions

real **abs**(real *x*)

>The absolute value of *x*. This function is deprecated and will be removed in the future; please use fabs instead.

real **fabs**(real *x*)

>The absolute value of *x*; see warning at start of Section 29.5

real **fdim**(real *x*, real *y*)

>The positive difference between *x* and *y*, which is *x* - *y* if *x* is greater than *y* and 0 otherwise; see warning at start of Section 29.5

Bounds Functions

real **fmin**(real *x*, real *y*)

>The minimum of *x* and *y*; see warning at start of Section 29.5

real **fmax**(real *x*, real *y*)

>The maximum of *x* and *y*; see warning at start of Section 29.5

Arithmetic Functions

real **fmod**(real *x*, real *y*)

>The real value remainder after dividing *x* by *y*; see warning at start of Section 29.5

Rounding Functions

Warning: Rounding functions convert real values to integers. Because the output is an integer, any gradient information resulting from functions applied to the integer is not passed to the real value it was derived from. With MCMC sampling using HMC or NUTS, the Metropolis/slice procedure will correct for any error due to poor gradient calculations, but the result is likely to be reduced acceptance probabilities and less efficient sampling.

One case where rounding is reasonable is linear interpolation of function values. For example, consider

```
xm <- x[floor(c)];
xp <- x[ceil(c)];
x <- (c - floor(c))*xp + (ceil(c) - c)*xm;
```

In this example, the first derivative for `x` with respect to `c` will be reasonable, but the second derivatives will all be zero.

real **floor**(real *x*)

The floor of *x*, which is the largest integer less than or equal to *x*, converted to a real value; see warning at start of Section 29.5

real **ceil**(real *x*)

The ceiling of *x*, which is the smallest integer greater than or equal to *x*, converted to a real value; see warning at start of Section 29.5

real **round**(real *x*)

The nearest integer to *x*, converted to a real value; see warning at start of Section 29.5

real **trunc**(real *x*)

The integer nearest to but no larger in magnitude than *x*, converted to a double value; see warning at start of Section 29.5

29.6. Power and Logarithm Functions

real **sqrt**(real *x*)

The square root of *x*

real **cbrt**(real *x*)

The cube root of *x*

real **square**(real *x*)

The square of *x*

real **exp**(real *x*)

The natural exponential of *x*

real **exp2**(real *x*)

The base-2 exponential of *x*

real **log**(real *x*)

The natural logarithm of *x*

real **log2**(real *x*)

The base-2 logarithm of *x*

```
real log10(real x)
```
      The base-10 logarithm of *x*

```
real pow(real x, real y)
```
      x raised to the power of *y*

```
real inv(real x)
```
      The inverse of *x*

```
real inv_sqrt(real x)
```
      The inverse of the square root of *x*

```
real inv_square(real x)
```
      The inverse of the square of *x*

29.7. Trigonometric Functions

```
real hypot(real x, real y)
```
      The length of the hypotenuse of a right triangle with sides of length *x* and *y*

```
real cos(real x)
```
      The cosine of the angle *x* (in radians)

```
real sin(real x)
```
      The sine of the angle *x* (in radians)

```
real tan(real x)
```
      The tangent of the angle *x* (in radians)

```
real acos(real x)
```
      The principal arc (inverse) cosine (in radians) of *x*

```
real asin(real x)
```
      The principal arc (inverse) sine (in radians) of *x*

```
real atan(real x)
```
      The principal arc (inverse) tangent (in radians) of *x*

```
real atan2(real x, real y)
```
      The principal arc (inverse) tangent (in radians) of *x* divided by *y*

29.8. Hyperbolic Trigonometric Functions

real **cosh**(real *x*)

>   The hyperbolic cosine of *x* (in radians)

real **sinh**(real *x*)

>   The hyperbolic sine of *x* (in radians)

real **tanh**(real *x*)

>   The hyperbolic tangent of *x* (in radians)

real **acosh**(real *x*)

>   The inverse hyperbolic cosine (in radians) of *x*

real **asinh**(real *x*)

>   The inverse hyperbolic sine (in radians) of *x*

real **atanh**(real *x*)

>   The inverse hyperbolic tangent (in radians) of *x*

29.9. Link Functions

The following functions are commonly used as link functions in generalized linear models (see Section 12.4). The function Φ is also commonly used as a link function (see Section 29.10).

real **logit**(real *x*)

>   The log odds, or logit, function applied to *x*, defined by $\text{logit}(x) = \log\left(\frac{x}{1-x}\right)$.

real **inv_logit**(real *y*)

>   The logistic sigmoid function applied to *y*, defined by $\text{logit}^{-1}(y) = \frac{1}{1+\exp(-y)}$.

real **inv_cloglog**(real *y*)

>   The inverse of the complementary log-log function applied to *y*, defined by $\text{cloglog}^{-1}(y) = 1 - \exp\left(-\exp(y)\right)$.

29.10. Probability-Related Functions

real **erf**(real *x*)

>   The error function of *x*

real **erfc**(real *x*)

    The complementary error function of *x*

real **Phi**(real *x*)

    The cumulative unit normal density function of *x*; Phi(*x*) will underflow to 0 for *x* below -37.5 and overflow to 1 for *x* above 8.25; derivatives will underflow to 0 below -27.5 and overflow to 1 above 27.5.

real **Phi_approx**(real *x*)

    Fast approximation of the cumulative unit normal density function of *x*, defined by

$$\Phi_{\text{approx}}(x) = \text{logit}^{-1}(0.07056\,x^3 + 1.5976\,x).$$

    This approximation has a maximum absolute error of 0.00014 and may be used instead of Phi for probit regression. See (Bowling et al., 2009) for details.

real **binary_log_loss**(int *y*, real *y_hat*)

    The log loss of predicting probability *y_hat* for binary outcome *y*;

    The log loss function for predicting $\hat{y} \in [0, 1]$ for boolean outcome $y \in \{0, 1\}$ is defined by

$$\texttt{binary\_log\_loss}(y, \hat{y}) = y \log \hat{y} + (1 - y) \log(1 - \hat{y}).$$

real **owens_t**(real *h*, real *a*)

    The Owen's T function for the probability of the event $X > a$ and $0 < Y < aX$ where X and Y are independent standard normal random variables, defined by

$$\text{owens\_t}(h, a) = \frac{1}{2\pi} \int_0^a \frac{\exp(-\frac{1}{2}h^2(1 + x^2))}{1 + x^2} dx$$

29.11. Combinatorial Functions

real **lbeta**(real *alpha*, real *beta*)

    The natural logarithm of the beta function applied to *alpha* and *beta*. The beta function, $B(\alpha, \beta)$, computes the normalizing constant for the beta distribution, and is defined for $\alpha > 0$ and $\beta > 0$ by

$$B(\alpha, \beta) = \int_0^1 u^{\alpha-1}(1 - u)^{\beta-1} du.$$

real **tgamma**(real *x*)

The gamma function applied to *x*. The gamma function is the generalization of the factorial function to continuous variables, defined so that $\Gamma(n+1) = n!$. The function is defined for positive numbers and non-integral negative numbers by

$$\Gamma(x) = \int_0^\infty u^{x-1} \exp(-u) \, du.$$

real **lgamma**(real *x*)

The natural logarithm of the gamma function applied to *x*

real **digamma**(real *x*)

The digamma function applied to *x*. The digamma function is the derivative of the natural logarithm of the Gamma function. The function is defined for positive numbers and non-integral negative numbers.

real **trigamma**(real *x*)

The trigamma function applied to *x*. The trigamma function is the second derivative of the natural logarithm of the Gamma function.

real **lmgamma**(int *n*, real *x*)

The natural logarithm of the multinomial gamma function with *n* dimensions applied to *x*.

real **gamma_p**(real *a*, real *z*)

The normalised lower incomplete gamma function of *a* and *z* defined for positive *a* and nonnegative *z* by

$$P(a, z) = \frac{1}{\Gamma(a)} \int_0^z t^{a-1} e^{-t} dt$$

real **gamma_q**(real *a*, real *z*)

The normalised upper incomplete gamma function of *a* and *z* defined for positive *a* and nonnegative *z* by

$$Q(a, z) = \frac{1}{\Gamma(a)} \int_z^\infty t^{a-1} e^{-t} dt$$

real **binomial_coefficient_log**(real *x*, real *y*)

The natural logarithm of the binomial coefficient of *x* and *y*. For non-negative integer inputs, this is pronounced "*x* choose *y*," written as $\binom{x}{y}$, and defined by

$$\binom{x}{y} = \frac{x!}{(x-y)!\,y!}.$$

This Stan function extends the domain to continuous quantities through the Γ function, defining

$$\text{binomial\_coefficient\_log}(x, y) = \log\frac{\Gamma(x+1)}{\Gamma(x-y+1)\,\Gamma(y+1)}.$$

real **bessel_first_kind**(int *v*, real *z*)

The Bessel function of the first kind with order *v* applied to *z* defined for all *z* and *v* by

$$J_v(z) = \left(\frac{1}{2}z\right)^v \sum_{k=0}^{\infty} \frac{\left(-\frac{1}{4}z^2\right)^k}{k!\,\Gamma(v+k+1)}$$

real **bessel_second_kind**(int *v*, real *z*)

The Bessel function of the second kind with order *v* applied to *z* defined for positive *z* and *v* by

$$Y_v(z) = \frac{J_v(z)\cos(v\pi) - J_{-v}(z)}{\sin(v\pi)}$$

real **modified_bessel_first_kind**(int *v*, real *z*)

The modified Bessel function of the first kind with order *v* applied to *z* defined for all *z* and *v* by

$$I_v(z) = \left(\frac{1}{2}z\right)^v \sum_{k=0}^{\infty} \frac{\left(\frac{1}{4}z^2\right)^k}{k!\,\Gamma(v+k+1)}$$

real **modified_bessel_second_kind**(int *v*, real *z*)

The modified Bessel function of the second kind with order *v* applied to *z* defined for positive *z* and *v* by

$$K_v(z) = \frac{\pi}{2} \cdot \frac{I_{-v}(z) - I_v(z)}{\sin(v\pi)}$$

real **falling_factorial**(real *x*, real *n*)

    The falling factorial of *x* with power *n* defined for positive *x* and real *n* by

$$\text{falling\_factorial}(x, n) = \frac{x!}{n!}$$

real **log_falling_factorial**(real *x*, real *n*)

    The log of the falling factorial of *x* with power *n* defined for positive *x* and real *n*.

real **rising_factorial**(real *x*, real *n*)

    The rising factorial of *x* with power *n* defined for positive *x* and real *n* by

$$\text{rising\_factorial}(x, n) = \frac{(x + n - 1)!}{(x - 1)!}$$

real **log_rising_factorial**(real *x*, real *n*)

    The log of the rising factorial of *x* with power *n* defined for positive *x* and real *n*.

29.12. Composed Functions

The functions in this section are equivalent in theory to combinations of other functions. In practice, they are implemented to be more efficient and more numerically stable than defining them directly using more basic Stan functions.

real **expm1**(real *x*)

    The natural exponential of *x* minus 1

real **fma**(real *x*, real *y*, real *z*)

    z plus the result of *x* multiplied by *y*

real **multiply_log**(real *x*, real *y*)

    The product of *x* and the natural logarithm of *y*; if both *x* and *y* are 0, the return value is 0

real **log1p**(real *x*)

    The natural logarithm of 1 plus *x*

real **log1m**(real *x*)

    The natural logarithm of 1 minus *x*

real **log1p_exp**(real *x*)

    The natural logarithm of one plus the natural exponentiation of *x*

real **log1m_exp**(real *x*)

    The natural logarithm of one minus the natural exponentiation of *x*.

real **log_diff_exp**(real *x*, real *y*)

    The natural logarithm of the difference of the natural exponentiation of *x* and the natural exponentiation of *y*

real **log_sum_exp**(real *x*, real *y*)

    The natural logarithm of the sum of the natural exponentiation of *x* and the natural exponentiation of *y*

real **log_inv_logit**(real *x*)

    The natural logarithm of the inverse logit function of *x*

real **log1m_inv_logit**(real *x*)

    The natural logarithm of 1 minus the inverse logit function of *x*

30.   Array Operations

30.1.   Reductions

The following operations take arrays as input and produce single output values.

Minimum and Maximum

```
real min(real x[])
```
> The minimum value in x, or $+\infty$ if x is empty

```
int min(int x[])
```
> The minimum value in x, or raise exception if x is empty

```
real max(real x[])
```
> The maximum value in x, or $-\infty$ if x is empty

```
int max(int x[])
```
> The maximum value in x, or raise exception if x is empty

Sum, Product, and Log Sum of Exp

```
int sum(int x[])
```
> The sum of the elements in x, or 0 if x is empty.

```
real sum(real x[])
```
> The sum of the elements in x, or 0 if x is empty.

```
real prod(real x[])
```
> The product of the elements in x, or 1 if x is empty.

```
real prod(int x[])
```
> The product of the elements in x, or 1 if x is empty.

```
real log_sum_exp(real x[])
```
> The natural logarithm of the sum of the exponentials of the elements in x

Moments

Moments are only defined for arrays x of size $N \geq 1$; it is an error to call them with arrays of size $N = 0$. The sample mean is defined by

$$\text{mean}(x) = \frac{1}{N} \sum_{n=1}^{N} x_n.$$

For $N \geq 2$, sample variance is defined for $N \geq 1$ by

$$\text{variance}(x) = \frac{1}{N-1} \sum_{n=1}^{N} (x_n - \text{mean}(x))^2$$

and sample standard deviation by

$$\text{sd}(x) = \sqrt{\text{variance}(x)},$$

For convenience, when $N = 1$, variance(x) and sd(x) are defined to be 0.

real **mean**(real *x*[])
> The sample mean of the elements in *x*

real **variance**(real *x*[])
> The sample variance of the elements in *x*

real **sd**(real *x*[])
> The sample standard deviation of elements in *x*

30.2. Array Size and Dimension Function

The size of an array or matrix can be obtained using the `dims()` function. The `dims()` function is defined to take an argument consisting of any variable with up to 8 array dimensions (and up to 2 additional matrix dimensions) and returns an array of integers with the dimensions. For example, if two variables are declared as follows,

```
real x[7,8,9];
matrix[8,9] y[7];
```

then calling `dims(x)` or `dims(y)` returns an integer array of size 3 containing the elements 7, 8, and 9 in that order.

The `size()` function extracts the number of elements in an array. The function is overloaded to apply to arrays of integers, reals, matrices, vectors, and row vectors.

```
int[] dims(T x)
```
> Returns an integer array containing the dimensions of *x*; the type of the
> argument *T* can be any Stan type with up to 8 array dimensions.

```
int size(T[] x)
```
> Returns the number of elements in the array *x*; the type of the array *T* can be
> anything type.

30.3. Array Broadcasting

The following operations create arrays by repeating elements to fill an array of a
specified size. These operations work for all input types *T*, including reals, integers,
vectors, row vectors, matrices, or arrays.

```
T[] rep_array(T x, int n)
```
> Return the *n* array with every entry assigned to *x*.

```
T[,] rep_array(T x, int m, int n)
```
> Return the *m* by *n* array with every entry assigned to *x*.

```
T[,,] rep_array(T x, int k, int m, int n)
```
> Return the *k* by *m* by *n* array with every entry assigned to *x*.

For example, `rep_array(1.0,5)` produces a real array (type `real[]`) of size 5 with
all values set to 1.0. On the other hand, `rep_array(1,5)` produces an integer array
(type `int[]`) of size 5 with all values set to 1. This distinction is important because
it is not possible to assign an integer array to a real array. For example, the following
example contrasts legal with illegal array creation and assignment

```
    real y[5];
    int x[5];

    x <- rep_array(1,5);     // ok
    y <- rep_array(1.0,5);   // ok

    x <- rep_array(1.0,5);   // illegal
    y <- rep_array(1,5);     // illegal

    x <- y;                  // illegal
    y <- x;                  // illegal
```

If the value being repeated v is a vector (i.e., T is `vector`), then `rep_array(v,27)`
is a size 27 array consisting of 27 copies of the vector v.

```
vector[5] v;
vector[5] a[3];
...
a <- rep_array(v,3);  // fill a with copies of v
a[2,4] <- 9.0;        // v[4], a[1,4], a[2,4] unchanged
```

   If the type *T* of *x* is itself an array type, then the result will be an array with one, two, or three added dimensions, depending on which of the `rep_array` functions is called. For instance, consider the following legal code snippet.

```
real a[5,6];
real b[3,4,5,6];
...
b <- rep_array(a,3,4); //  make (3 x 4) copies of a
b[1,1,1,1] <- 27.9;    //  a[1,1] unchanged
```

After the assignment to b, the value for b[j,k,m,n] is equal to a[m,n] where it is defined, for j in 1:3, k in 1:4, m in 1:5, and n in 1:6.

30.4. Other functions

real[] **sort_asc**(real[] *v*)

   Sort the elements of *v* in ascending order

int[] **sort_asc**(int[] *v*)

   Sort the elements of *v* in ascending order

real[] **sort_desc**(real[] *v*)

   Sort the elements of *v* in descending order

int[] **sort_desc**(int[] *v*)

   Sort the elements of *v* in descending order

int **rank**(real[] *v*, int *s*)

   Number of components of *v* less than *v[s]*

int **rank**(int[] *v*, int *s*)

   Number of components of *v* less than *v[s]*

31.   Matrix Operations

31.1.   Integer-Valued Matrix Size Functions

int **rows**(vector *x*)
>   The number of rows in the vector *x*

int **rows**(row_vector *x*)
>   The number of rows in the row vector *x*, namely 1

int **rows**(matrix *x*)
>   The number of rows in the matrix *x*

int **cols**(vector *x*)
>   The number of columns in the vector *x*, namely 1

int **cols**(row_vector *x*)
>   The number of columns in the row vector *x*

int **cols**(matrix *x*)
>   The number of columns in the matrix *x*

31.2.   Matrix Arithmetic Operators

Stan supports the basic matrix operations using infix, prefix and postfix operations. This section lists the operations supported by Stan along with their argument and result types.

Negation Prefix Operators

vector **operator-**(vector *x*)
>   The negation of the vector *x*

row_vector **operator-**(row_vector *x*)
>   The negation of the row vector *x*

matrix **operator-**(matrix *x*)
>   The negation of the matrix *x*

Infix Matrix Operators

```
vector operator+(vector x, vector y)
```
    The sum of the vectors *x* and *y*

```
row_vector operator+(row_vector x, row_vector y)
```
    The sum of the row vectors *x* and *y*

```
matrix operator+(matrix x, matrix y)
```
    The sum of the matrices *x* and *y*

```
vector operator-(vector x, vector y)
```
    The difference between the vectors *x* and *y*

```
row_vector operator-(row_vector x, row_vector y)
```
    The difference between the row vectors *x* and *y*

```
matrix operator-(matrix x, matrix y)
```
    The difference between the matrices *x* and *y*

```
vector operator*(real x, vector y)
```
    The product of the scalar *x* and vector *y*

```
row_vector operator*(real x, row_vector y)
```
    The product of the scalar *x* and the row vector *y*

```
matrix operator*(real x, matrix y)
```
    The product of the scalar *x* and the matrix *y*

```
vector operator*(vector x, real y)
```
    The product of the scalar *y* and vector *x*

```
matrix operator*(vector x, row_vector y)
```
    The product of the vector *x* and row vector *y*

```
row_vector operator*(row_vector x, real y)
```
    The product of the scalar *y* and row vector *x*

```
real operator*(row_vector x, vector y)
```
    The product of the row vector *x* and vector *y*

```
row_vector operator*(row_vector x, matrix y)
```
    The product of the row vector *x* and matrix *y*

```
matrix operator*(matrix x, real y)
```
      The product of the scalar *y* and matrix *x*

```
vector operator*(matrix x, vector y)
```
      The product of the matrix *x* and vector *y*

```
matrix operator*(matrix x, matrix y)
```
      The product of the matrices *x* and *y*

Broadcast Infix Operators

```
vector operator+(vector x, real y)
```
      The result of adding *y* to every entry in the vector *x*

```
vector operator+(real x, vector y)
```
      The result of adding *x* to every entry in the vector *y*

```
row_vector operator+(row_vector x, real y)
```
      The result of adding *y* to every entry in the row vector *x*

```
row_vector operator+(real x, row_vector y)
```
      The result of adding *x* to every entry in the row vector *y*

```
matrix operator+(matrix x, real y)
```
      The result of adding *y* to every entry in the matrix *x*

```
matrix operator+(real x, matrix y)
```
      The result of adding *x* to every entry in the matrix *y*

```
vector operator-(vector x, real y)
```
      The result of subtracting *y* from every entry in the vector *x*

```
vector operator-(real x, vector y)
```
      The result of adding *x* to every entry in the negation of the vector *y*

```
row_vector operator-(row_vector x, real y)
```
      The result of subtracting *y* from every entry in the row vector *x*

```
row_vector operator-(real x, row_vector y)
```
      The result of adding *x* to every entry in the negation of the row vector *y*

```
matrix operator-(matrix x, real y)
```
      The result of subtracting *y* from every entry in the matrix *x*

```
matrix operator-(real x, matrix y)
```
   The result of adding *x* to every entry in negation of the matrix *y*

```
vector operator/(vector x, real y)
```
   The result of dividing each entry in the vector *x* by *y*

```
row_vector operator/(row_vector x, real y)
```
   The result of dividing each entry in the row vector *x* by *y*

```
matrix operator/(matrix x, real y)
```
   The result of dividing each entry in the matrix *x* by *y*

Elementwise Products

```
vector operator.*(vector x, vector y)
```
   The elementwise product of *y* and *x*

```
row_vector operator.*(row_vector x, row_vector y)
```
   The elementwise product of *y* and *x*

```
matrix operator.*(matrix x, matrix y)
```
   The elementwise product of *y* and *x*

```
vector operator./(vector x, vector y)
```
   The elementwise quotient of *y* and *x*

```
row_vector operator./(row_vector x, row_vector y)
```
   The elementwise quotient of *y* and *x*

```
matrix operator./(matrix x, matrix y)
```
   The elementwise quotient of *y* and *x*

Elementwise Logarithms

```
vector log(vector x)
```
   The elementwise natural logarithm of *x*

```
row_vector log(row_vector x)
```
   The elementwise natural logarithm of *x*

```
matrix log(matrix x)
```
The elementwise natural logarithm of *x*

```
vector exp(vector x)
```
The elementwise exponential of *x*

```
row_vector exp(row_vector x)
```
The elementwise exponential of *x*

```
matrix exp(matrix x)
```
The elementwise exponential of *x*

Cumulative Sums

The cumulative sum of a sequence $x_1, \ldots, x_N$ is the sequence $y_1, \ldots, y_N$, where

$$y_n = \sum_{m=1}^{n} x_n.$$

```
real[] cumulative_sum(real[] x)
```
The cumulative sum of *x*

```
vector cumulative_sum(vector v)
```
The cumulative sum of *v*

```
row_vector cumulative_sum(row_vector rv)
```
The cumulative sum of *rv*

Dot Products

```
real dot_product(vector x, vector y)
```
The dot product of *x* and *y*

```
real dot_product(vector x, row_vector y)
```
The dot product of *x* and *y*

```
real dot_product(row_vector x, vector y)
```
The dot product of *x* and *y*

```
real dot_product(row_vector x, row_vector y)
```
The dot product of *x* and *y*

row_vector **columns_dot_product**(vector *x*, vector *y*)
    The dot product of the columns of *x* and *y*

row_vector **columns_dot_product**(row_vector *x*, row_vector *y*)
    The dot product of the columns of *x* and *y*

row_vector **columns_dot_product**(matrix *x*, matrix *y*)
    The dot product of the columns of *x* and *y*

vector **rows_dot_product**(vector *x*, vector *y*)
    The dot product of the rows of *x* and *y*

vector **rows_dot_product**(row_vector *x*, row_vector *y*)
    The dot product of the rows of *x* and *y*

vector **rows_dot_product**(matrix *x*, matrix *y*)
    The dot product of the rows of *x* and *y*

real **dot_self**(vector *x*)
    The dot product of the vector *x* with itself

real **dot_self**(row_vector *x*)
    The dot product of the row vector *x* with itself

row_vector **columns_dot_self**(vector *x*)
    The dot product of the columns of *x* with themselves

row_vector **columns_dot_self**(row_vector *x*)
    The dot product of the columns of *x* with themselves

row_vector **columns_dot_self**(matrix *x*)
    The dot product of the columns of *x* with themselves

vector **rows_dot_self**(vector *x*)
    The dot product of the rows of *x* with themselves

vector **rows_dot_self**(row_vector *x*)
    The dot product of the rows of *x* with themselves

vector **rows_dot_self**(matrix *x*)
    The dot product of the rows of *x* with themselves

Specialized Products

matrix **tcrossprod**(matrix *x*)

    The product of *x* postmultiplied by its own transpose, similar to the tcrossprod(x) function in R. The result is a symmetric matrix $x\,x^\top$.

matrix **crossprod**(matrix *x*)

    The product of *x* premultiplied by its own transpose, similar to the crossprod(x) function in R. The result is a symmetric matrix $x^\top x$.

    The following functions all provide shorthand forms for common expressions, which are also much more efficient.

matrix **quad_form**(matrix *A*, matrix *B*)

    The quadratic form, i.e., B' * A * B.

real **quad_form**(matrix *A*, vector *B*)

    The quadratic form, i.e., B' * A * B.

real **trace_quad_form**(matrix *A*, matrix *B*)

    The trace of the quadratic form, i.e., trace(B' * A * B).

real **trace_gen_quad_form**(matrix *D*,matrix *A*, matrix *B*)

    The trace of a generalized quadratic form, i.e., trace(D * B' * A * B).

matrix **multiply_lower_tri_self_transpose**(matrix *x*)

    The product of the lower triangular portion of *x* (including the diagonal) times its own transpose; that is, if L is a matrix of the same dimensions as *x* with L(m,n) equal to *x*(m,n) for n ≤ m and L(m,n) equal to 0 if n > m, the result is the symmetric matrix $L\,L^\top$. This is a specialization of tcrossprod(x) for lower-triangular matrices.

matrix **diag_pre_multiply**(vector *v*, matrix *m*)

    Return the product of the diagonal matrix formed from the vector *v* and the matrix *m*, i.e., diag_matrix(*v*) * *m*.

matrix **diag_pre_multiply**(row_vector *rv*, matrix *m*)

    Return the product of the diagonal matrix formed from the vector *rv* and the matrix *m*, i.e., diag_matrix(*rv*) * *m*.

matrix **diag_post_multiply**(matrix *m*, vector *v*)

    Return the product of the matrix *m* and the diagonal matrix formed from the vector *v*, i.e., *m* * diag_matrix(*v*).

```
matrix diag_post_multiply(matrix m, row_vector rv)
```
Return the product of the matrix *m* and the diagonal matrix fromed from the
the row vector *rv*, i.e., `m * diag_matrix(rv)`.

31.3.   Reductions

Log Sum of Exponents

```
real log_sum_exp(vector x)
```
The natural logarithm of the sum of the exponentials of the elements in *x*

```
real log_sum_exp(row_vector x)
```
The natural logarithm of the sum of the exponentials of the elements in *x*

```
real log_sum_exp(matrix x)
```
The natural logarithm of the sum of the exponentials of the elements in *x*

Minimum and Maximum

```
real min(vector x)
```
The minimum value in *x*, or $+\infty$ if *x* is empty

```
real min(row_vector x)
```
The minimum value in *x*, or $+\infty$ if *x* is empty

```
real min(matrix x)
```
The minimum value in *x*, or $+\infty$ if *x* is empty

```
real max(vector x)
```
The maximum value in *x*, or $-\infty$ if *x* is empty

```
real max(row_vector x)
```
The maximum value in *x*, or $-\infty$ if *x* is empty

```
real max(matrix x)
```
The maximum value in *x*, or $-\infty$ if *x* is empty

Sums and Products

```
real sum(vector x)
```
The sum of the values in *x*, or 0 if *x* is empty

real **sum**(row_vector x)

    The sum of the values in x, or 0 if x is empty

real **sum**(matrix x)

    The sum of the values in x, or 0 if x is empty

real **prod**(vector x)

    The product of the values in x, or 1 if x is empty

real **prod**(row_vector x)

    The product of the values in x, or 1 if x is empty

real **prod**(matrix x)

    The product of the values in x, or 1 if x is empty

Sample Moments

Full definitions are provided for sample moments in Section 30.1.

real **mean**(vector x)

    The sample mean of the values in x; see Section 30.1 for details.

real **mean**(row_vector x)

    The sample mean of the values in x; see Section 30.1 for details.

real **mean**(matrix x)

    The sample mean of the values in x; see Section 30.1 for details.

real **variance**(vector x)

    The sample variance of the values in x; see Section 30.1 for details.

real **variance**(row_vector x)

    The sample variance of the values in x; see Section 30.1 for details.

real **variance**(matrix x)

    The sample variance of the values in x; see Section 30.1 for details.

real **sd**(vector x)

    The sample standard deviation of the values in x; see Section 30.1 for details.

real **sd**(row_vector x)

    The sample standard deviation of the values in x; see Section 30.1 for details.

```
real sd(matrix x)
```
The sample standard deviation of the values in *x*; see Section 30.1 for details.

31.4.   Broadcast Functions

The following broadcast functions allow vectors, row vectors and matrices to be created by copying a single element into all of their cells. Matrices may also be created by stacking copies of row vectors vertically or stacking copies of column vectors horizontally.

```
vector rep_vector(real x, int m)
```
Return the size *m* (column) vector consisting of copies of *x*.

```
row_vector rep_row_vector(real x, int n)
```
Return the size *n* row vector consisting of copies of *x*.

```
matrix rep_matrix(real x, int m, int n)
```
Return the *m* by *n* matrix consisting of copies of *x*.

```
matrix rep_matrix(vector v, int n)
```
Return the *m* by *n* matrix consisting of *n* copies of the (column) vector *v* of size *m*.

```
matrix rep_matrix(row_vector rv, int m)
```
Return the *m* by *n* matrix consisting of *m* copies of the row vector *rv* of size *n*.

Unlike the situation with array broadcasting (see Section 30.3), where there is a distinction between integer and real arguments, the following two statements produce the same result for vector broadcasting; row vector and matrix broadcasting behave similarly.

```
vector[3] x;
x <- rep_vector(1, 3);
x <- rep_vector(1.0, 3);
```

There are no integer vector or matrix types, so integer values are automatically promoted.

```
vector to_vector(matrix m)
```
Return the matrix *m* as a vector in column-major order.

```
vector to_vector(row_vector m)
```
Return the row vector *m* as a vector in column-major order.

31.5.   Slice and Package Functions

Diagonal Matrices

vector **diagonal**(matrix *x*)

> The diagonal of the matrix *x*

matrix **diag_matrix**(vector *x*)

> The diagonal matrix with diagonal *x*

Columns and Rows

vector **col**(matrix *x*, int *n*)

> The *n*-th column of matrix *x*

row_vector **row**(matrix *x*, int *m*)

> The *m*-th row of matrix *x*

Block Operations

Matrix Slicing Operations

Block operations may be used to extract a sub-block of a matrix.

matrix **block**(matrix *x*, int *i*, int *j*, int *n_rows*, int *n_cols*)

> Return the submatrix of *x* that starts at row *i* and column *j* and extends *n_rows* rows and *n_cols* columns.

The sub-row and sub-column operations may be used to extract a slice of row or column from a matrix

vector **sub_col**(matrix *x*, int *i*, int *j*, int *n_rows*)

> Return the sub-column of *x* that starts at row *i* and column *j* and extends *n_rows* rows and 1 column.

row_vector **sub_row**(matrix *x*, int *i*, int *j*, int *n_cols*)

> Return the sub-row of *x* that starts at row *i* and column *j* and extends 1 row and *n_cols* columns.

Vector and Array Slicing Operations

The head operation extracts the first n elements of a vector and the tail operation the last. The segment operation extracts an arbitrary subvector.

vector **head**(vector *v*, int *n*)

Return the vector consisting of the first n elements of *v*.

row_vector **head**(row_vector *rv*, int *n*)

Return the row vector consisting of the first n elements of *rv*.

T[] **head**(T[] *sv*, int *n*)

Return the standard vector consisting of the first n elements of *sv*; applies to up to three-dimensional arrays containing any type of elements T.

vector **tail**(vector *v*, int *n*)

Return the vector consisting of the last n elements of *v*.

row_vector **tail**(row_vector *rv*, int *n*)

Return the row vector consisting of the last n elements of *rv*.

T[] **tail**(T[] *sv*, int *n*)

Return the standard vector consisting of the last n elements of *sv*; applies to up to three-dimensional arrays containing any type of elements T.

vector **segment**(vector *v*, int *i*, int *n*)

Return the vector consisting of the n elements of *v* starting at i; i.e., elements i through through $i + n - 1$.

row_vector **segment**(row_vector *v*, int *i*, int *n*)

Return the row vector consisting of the n elements of *rv* starting at i; i.e., elements i through through $i + n - 1$.

T[] **segment**(T[] *sv*, int *i*, int *n*)

Return the standard vector consisting of the n elements of *sv* starting at i; i.e., elements i through through $i + n - 1$. Applies to up to three-dimensional arrays containing any type of elements T.

Transposition Postfix Operator

matrix **operator'**(matrix *x*)

The transpose of the matrix *x*, written as x'

```
row_vector operator'(vector x)
```
The transpose of the vector x, written as x'

```
vector operator'(row_vector x)
```
The transpose of the vector x, written as x'

31.6. Special Matrix Functions

The softmax function maps $y \in \mathbb{R}^K$ to the K-simplex by

$$\text{softmax}(y) = \frac{\exp(y)}{\sum_{k=1}^{K} \exp(y_k)},$$

where $\exp(y)$ is the componentwise exponentiation of y. Softmax is usually calculated on the log scale,

$$\log \text{softmax}(y) \;=\; y - \log \sum_{k=1}^{K} \exp(y_k) \;=\; y - \text{log\_sum\_exp}(y).$$

The entries in the Jacobian of the softmax function are given by

$$\frac{\partial}{\partial y_m} \text{softmax}(y)[k]$$

$$= \begin{cases} \text{softmax}(y)[k] - \text{softmax}(y)[k] \times \text{softmax}(y)[m] & \text{if } m = k, \text{ and} \\ \text{softmax}(y)[k] * \text{softmax}(y)[m] & \text{if } m \neq k. \end{cases}$$

For the log softmax function, the entries are

$$\frac{\partial}{\partial y_m} \text{softmax}(y)[k] = \begin{cases} 1 - \text{softmax}(y)[m] & \text{if } m = k, \text{ and} \\ \text{softmax}(y)[m] & \text{if } m \neq k. \end{cases}$$

Stan provides the following functions for softmax and its log.

```
vector softmax(vector x)
```
The softmax of x

```
vector log_softmax(vector x)
```
The natural logarithm of the softmax of x

31.7. Linear Algebra Functions and Solvers

Matrix Division Infix Operators

row_vector **operator/**(row_vector *b*, matrix *A*)

> The right division of *b* by *A*; equivalently *b* * inverse(*A*)

matrix **operator/**(matrix *b*, matrix *A*)

> The right division of *b* by *A*; equivalently *b* * inverse(*A*)

vector **operator**(matrix *A*, vector *b*)

> T he left division of *b* by *A*; equivalently inverse(*A*) * *b*

matrix **operator**(matrix *A*, matrix *b*)

> T he left division of *b* by *A*; equivalently inverse(*A*) * *b*

Lower-Triangular Matrix-Division Functions

There are four division functions which use lower triangular views of a matrix. The lower triangular view of a matrix tri(A) is defined by

$$\mathrm{tri}(A)[m, n] = \begin{cases} A[m, n] & \text{if } m \geq n, \text{ and} \\ 0 & \text{otherwise.} \end{cases}$$

row_vector **mdivide_right_tri_low**(row_vector *b*, matrix *a*)

> The right division of *b* by tri(*a*), a lower triangular view of *a*; equivalently *b* * inverse(tri(*a*))

matrix **mdivide_right_tri_low**(matrix *b*, matrix *a*)

> The right division of *b* by tri(*a*), a lower triangular view of *a*; equivalently *b* * inverse(tri(*a*))

vector **mdivide_left_tri_low**(matrix *a*, vector *b*)

> T he left division of *b* by a triangular view of tri(*a*), a lower triangular view of *a*; equivalently inverse(tri(*a*)) * *b*

matrix **mdivide_left_tri_low**(matrix *a*, matrix *b*)

> T he left division of *b* by a triangular view of tri(*a*), a lower triangular view of *a*; equivalently inverse(tri(*a*)) * *b*

Linear Algebra Functions

real **trace**(matrix *A*)

    The trace of *A*, or 0 if *A* is empty; *A* is not required to be diagonal

real **determinant**(matrix *A*)

    The determinant of *A*

real **log_determinant**(matrix *A*)

    The log of the absolute value of the determinant of *A*

matrix **inverse**(matrix *A*)

    The inverse of *A*

matrix **inverse_spd**(matrix *A*)

    The inverse of *A* where A is symmetric, positive definite

vector **eigenvalues_sym**(matrix *A*)

    The vector of eigenvalues of a symmetric matrix *A* in descending order

matrix **eigenvectors_sym**(matrix *A*)

    The matrix with the eigenvectors of symmetric matrix *A*

matrix **cholesky_decompose**(matrix *A*)

    The lower-triangular Cholesky factor of *A*

vector **singular_values**(matrix *A*)

    The singular values of *A* in descending order

31.8.   Other functions

vector **sort_asc**(vector *v*)

    Sort the elements of *v* in ascending order

row_vector **sort_asc**(row_vector *v*)

    Sort the elements of *v* in ascending order

vector **sort_desc**(vector *v*)

    Sort the elements of *v* in descending order

row_vector **sort_desc**(row_vector *v*)

    Sort the elements of *v* in descending order

int **rank**(vector *v*, int *s*)

    Number of components of *v* less than *v[s]*

int **rank**(row_vector *v*, int *s*)

    Number of components of *v* less than *v[s]*

Part VI

Discrete Distributions

32.   Binary Distributions

Binary probability distributions have support on $\{0, 1\}$, where 1 represents the value true and 0 the value false.

32.1.   Bernoulli Distribution

Probability Mass Function

If $\theta \in [0, 1]$, then for $y \in \{0, 1\}$,

$$\text{Bernoulli}(y|\theta) = \left\{ \begin{array}{ll} \theta & \text{if } y = 1, \text{ and} \\ 1 - \theta & \text{if } y = 0. \end{array} \right.$$

Sampling Statement

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

Increment log probability with `bernoulli_log(`*`y`*`,`*`theta`*`)`, dropping constant additive terms; Section 23.3 explains sampling statements.

Stan Functions

`real `**`bernoulli_log`**`(ints `*`y`*`, reals `*`theta`*`)`

The log Bernoulli probability mass of *y* given chance of success *theta*

`real `**`bernoulli_cdf`**`(ints `*`y`*`, reals `*`theta`*`)`

The Bernoulli cumulative distribution function of *y* given chance of success *theta*

`real `**`bernoulli_cdf_log`**`(ints `*`y`*`, reals `*`theta`*`)`

The log of the Bernoulli cumulative distribution function of *y* given chance of success *theta*

`real `**`bernoulli_ccdf_log`**`(ints `*`y`*`, reals `*`theta`*`)`

The log of the Bernoulli complementary cumulative distribution function of *y* given chance of success *theta*

`int `**`bernoulli_rng`**`(real `*`theta`*`)`

Generate a Bernoulli variate with chance of success *theta*; may only be used in generated quantities block

32.2.   Bernoulli Distribution, Logit Parameterization

Stan also supplies a direct parameterization in terms of a logit-transformed chance-of-success parameter. This parameterization is more numerically stable if the chance-of-success parameter is on the logit scale, as with the linear predictor in a logistic regression.

Probability Mass Function

If $\alpha \in \mathbb{R}$, then for $c \in \{0, 1\}$,

$$\mathsf{BernoulliLogit}(c|\alpha) = \mathsf{Bernoulli}(c|\mathrm{logit}^{-1}(\alpha)) = \begin{cases} \mathrm{logit}^{-1}(\alpha) & \text{if } y = 1, \text{ and} \\ 1 - \mathrm{logit}^{-1}(\alpha) & \text{if } y = 0. \end{cases}$$

Sampling Statement

`y ~` **`bernoulli_logit`**`(alpha);`
> Increment log probability with `bernoulli_logit_log(y,alpha)`, dropping constant additive terms; Section 23.3 explains sampling statements.

Stan Functions

`real` **`bernoulli_logit_log`**`(ints y, reals alpha)`
> The log Bernoulli probability mass of y given logit chance of success `exp(alpha)`

33. Bounded Discrete Distributions

Bounded discrete probability functions have support on $\{0, \ldots, N\}$ for some upper bound N.

33.1. Binomial Distribution

Probability Mass Function

Suppose $N \in \mathbb{N}$ and $\theta \in [0,1]$, and $n \in \{0, \ldots, N\}$.

$$\text{Binomial}(n|N, \theta) = \binom{N}{n} \theta^n (1-\theta)^{N-n}.$$

Log Probability Mass Function

$$
\begin{aligned}
\log \text{Binomial}(n|N, \theta) \quad = \quad & \log \Gamma(N+1) - \log \Gamma(n+1) - \log \Gamma(N-n+1) \\
& + n \log \theta + (N-n) \log(1-\theta),
\end{aligned}
$$

Gradient of Log Probability Mass Function

$$\frac{\partial}{\partial \theta} \log \text{Binomial}(n|N, \theta) = \frac{n}{\theta} - \frac{N-n}{1-\theta}$$

Sampling Statement

`n ~ binomial(N, theta);`

Increment log probability with `binomial_log(n, N, theta)`, dropping constant additive terms; Section 23.3 explains sampling statements.

Stan Functions

real **binomial_log**(ints *n*, ints *N*, reals *theta*)

The log binomial probability mass of *n* successes in *N* trials given chance of success *theta*

real **binomial_cdf**(ints *n*, ints *N*, reals *theta*)

The binomial cumulative distribution function of *n* successes in *N* trials given chance of success *theta*

```
real binomial_cdf_log(ints n, ints N, reals theta)
```
The log of the binomial cumulative distribution function of *n* successes in *N* trials given chance of success *theta*

```
real binomial_ccdf_log(ints n, ints N, reals theta)
```
The log of the binomial complementary cumulative distribution function of *n* successes in *N* trials given chance of success *theta*

```
int binomial_rng(int N, real theta)
```
Generate a binomial variate with *N* trials and chance of success *theta*; may only be used in generated quantities block

33.2. Binomial Distribution, Logit Parameterization

Stan also provides a version of the binomial probability mass function distribution with the chance of success parameterized on the unconstrained logistic scale.

Probability Mass Function

Suppose $N \in \mathbb{N}$, $\alpha \in \mathbb{R}$, and $n \in \{0, \dots, N\}$.

$$
\begin{aligned}
\mathsf{BinomialLogit}(n|N, \alpha) &= \mathsf{BinomialLogit}(n|N, \mathrm{logit}^{-1}(\alpha)) \\
&= \binom{N}{n} \left(\mathrm{logit}^{-1}(\alpha)\right)^{n} \left(1 - \mathrm{logit}^{-1}(\alpha)\right)^{N-n}.
\end{aligned}
$$

Log Probability Mass Function

$$
\begin{aligned}
\log \mathsf{BinomialLogit}(n|N, \alpha) &= \log \Gamma(N+1) - \log \Gamma(n+1) - \log \Gamma(N-n+1) \\
&\quad + n \log \mathrm{logit}^{-1}(\alpha) + (N-n) \log \left(1 - \mathrm{logit}^{-1}(\alpha)\right),
\end{aligned}
$$

Gradient of Log Probability Mass Function

$$
\frac{\partial}{\partial \alpha} \log \mathsf{BinomialLogit}(n|N, \alpha) = \frac{n}{\mathrm{logit}^{-1}(-\alpha)} - \frac{N-n}{\mathrm{logit}^{-1}(\alpha)}
$$

 n ~ **binomial_logit**(*N,alpha*);
   Increment log probability with binomial_logit_log(*n,N,alpha*), dropping constant additive terms; Section 23.3 explains sampling statements.

Stan Functions

real **binomial_logit_log**(ints *n*, ints *N*, reals *alpha*)
   The log binomial probability mass of *n* successes in *N* trials given logit-scaled chance of success *alpha*

33.3. Beta-Binomial Distribution

Probability Mass Function

If $N \in \mathbb{N}$, $\alpha \in \mathbb{R}^+$, and $\beta \in \mathbb{R}^+$, then for $n \in \{0, \dots, N\}$,

$$\text{BetaBinomial}(n|N, \alpha, \beta) = \binom{N}{n} \frac{\text{B}(n + \alpha, N - n + \beta)}{\text{B}(\alpha, \beta)},$$

where the beta function $\text{B}(u, v)$ is defined for $u \in \mathbb{R}^+$ and $v \in \mathbb{R}^+$ by

$$\text{B}(u, v) = \frac{\Gamma(u)\,\Gamma(v)}{\Gamma(u + v)}.$$

Sampling Statement

 n ~ **beta_binomial**(*N,alpha,beta*);
   Increment log probability with beta_binomial_log(*n,N,alpha,beta*), dropping constant additive terms; Section 23.3 explains sampling statements.

Stan Functions

real **beta_binomial_log**(ints *n*, ints *N*, reals *alpha*, reals *beta*)
   The log beta-binomial probability mass of *n* successes in *N* trials given prior success count (plus one) of *alpha* and prior failure count (plus one) of *beta*

real **beta_binomial_cdf**(ints *n*, ints *N*, reals *alpha*, reals *beta*)
   The beta-binomial cumulative distribution function of *n* successes in *N* trials given prior success count (plus one) of *alpha* and prior failure count (plus one) of *beta*

real **beta_binomial_cdf_log**(ints *n*, ints *N*, reals *alpha*, reals *beta*)

>The log of the beta-binomial cumulative distribution function of *n* successes in *N* trials given prior success count (plus one) of *alpha* and prior failure count (plus one) of *beta*

real **beta_binomial_ccdf_log**(ints *n*, ints *N*, reals *alpha*, reals *beta*)

>The log of the beta-binomial complementary cumulative distribution function of *n* successes in *N* trials given prior success count (plus one) of *alpha* and prior failure count (plus one) of *beta*

int **beta_binomial_rng**(int *N*, real *alpha*, real *beta*)

>Generate a beta-binomial variate with *N* trials, prior success count (plus one) of *alpha*, and prior failure count (plus one) of *beta*; may only be used in generated quantities block

33.4. Hypergeometric Distribution

Probability Mass Function

If $a \in \mathbb{N}$, $b \in \mathbb{N}$, and $N \in \{0, \ldots, a+b\}$, then for $n \in \{\max(0, N-b), \ldots, \min(a, N)\}$,

$$\text{Hypergeometric}(n|N, a, b) = \frac{\binom{a}{n}\binom{b}{N-n}}{\binom{a+b}{N}}.$$

Sampling Statement

 n ~ **hypergeometric**(*N*, *a*, *b*);

>Increment log probability with hypergeometric_log(*n*, *N*, *a*, *b*), dropping constant additive terms; Section 23.3 explains sampling statements.

Stan Functions

real **hypergeometric_log**(int *n*, int *N*, int *a*, int *b*)

>The log hypergeometric probability mass of *n* successes in *N* trials given total success count of *a* and total failure count of *b*

int **hypergeometric_rng**(int *N*, real *a*, real *b*)

>Generate a hypergeometric variate with *N* trials, total success count of *a*, and total failure count of *b*; may only be used in generated quantities block

33.5. Categorical Distribution

Probability Mass Functions

If $N \in \mathbb{N}$, $N > 0$, and $\theta \in N$-simplex, then for $y \in \{1, \ldots, N\}$,

$$\mathsf{Categorical}(y|\theta) = \theta_y.$$

In addition, Stan provides a log-odds scaled categorical distribution,

$$\mathsf{CategoricalLogit}(y|\beta) = \mathsf{Categorical}(y|\mathrm{softmax}(\beta)).$$

See Section 31.6 for the definition of the softmax function.

Sampling Statement

`y ~ ` **`categorical`**`(theta);`

Increment log probability with `categorical_log(y,theta)`, dropping constant additive terms; Section 23.3 explains sampling statements.

Stan Functions

All of the categorical distributions are vectorized so that the outcome `y` can be a single integer (type `int`) or an array of integers (type `int[]`).

real **`categorical_log`**(ints *y*, vector *theta*)

The log categorical probability mass function with outcome(s) *y* in $1 : N$ given N-simplex distribution parameter *theta*.

real **`categorical_logit_log`**(ints *y*, vector *beta*)

The log categorical probability mass function with outcome(s) *y* in $1 : N$ given log-odds of outcomes *beta*.

int **`categorical_rng`**(vector *theta*)

Generate a categorical variate with N-simplex distribution parameter *theta*; may only be used in generated quantities block

33.6. Ordered Logistic Distribution

Probability Mass Function

If $K \in \mathbb{N}$ with $K > 2$, $c \in \mathbb{R}^{K-1}$ such that $c_k < c_{k+1}$ for $k \in \{1, \ldots, K-2\}$, and $\eta \in \mathbb{R}$, then for $k \in \{1, \ldots, K\}$,

$$
\text{OrderedLogistic}(k|\eta, c) = \left\{
\begin{array}{ll}
1 - \text{logit}^{-1}(\eta - c_1) & \text{if } k = 1, \\
\text{logit}^{-1}(\eta - c_{k-1}) - \text{logit}^{-1}(\eta - c_k) & \text{if } 1 < k < K, \text{and} \\
\text{logit}^{-1}(\eta - c_{K-1}) - 0 & \text{if } k = K.
\end{array}
\right.
$$

The $k = 1$ and $k = K$ edge cases can be subsumed into the general definition by setting $c_0 = -\infty$ and $c_K = +\infty$ with $\text{logit}^{-1}(-\infty) = 0$ and $\text{logit}^{-1}(\infty) = 1$.

Sampling Statement

`k ~ `**`ordered_logistic(`**`eta,c`**`);`**
> Increment log probability with `ordered_logistic_log(k,eta,c)`, dropping constant additive terms; Section 23.3 explains sampling statements.

Stan Functions

real **ordered_logistic_log**(int *k*, real *eta*, vector *c*)
> The log ordered logistic probability mass of *k* given linear predictor *eta* and cutpoints *c*.

int **ordered_logistic_rng**(real *eta*, vector *c*)
> Generate an ordered logistic variate with linear predictor *eta* and cutpoints *c*; may only be used in generated quantities block

34. Unbounded Discrete Distributions

The unbounded discrete distributions have support over the natural numbers (i.e., the non-negative integers).

34.1. Negative Binomial Distribution

Probability Mass Function

If $\alpha \in \mathbb{R}^+$ and $\beta \in \mathbb{R}^+$, then for $n \in \mathbb{N}$,

$$\mathsf{NegativeBinomial}(n|\alpha, \beta) = \binom{n + \alpha - 1}{\alpha - 1} \left(\frac{\beta}{\beta + 1}\right)^\alpha \left(\frac{1}{\beta + 1}\right)^n.$$

$$
\begin{aligned}
\log \mathsf{NegativeBinomial}(n|\alpha, \beta) \;=\;& \log \Gamma(n + \alpha) - \log \Gamma(n + 1) - \log \Gamma(\alpha) \\
&+ \alpha \left(\log \beta - \log(\beta + 1)\right) - n \log(\beta + 1)
\end{aligned}
$$

$$\frac{\partial}{\partial \alpha} \log \mathsf{NegativeBinomial}(n|\alpha, \beta) = \Psi(n + \alpha) - \Psi(\alpha) + \log \beta - \log(\beta + 1)$$

$$\frac{\partial}{\partial \beta} \log \mathsf{NegativeBinomial}(n|\alpha, \beta) = \frac{\alpha}{\beta} - \frac{\alpha + n}{\beta + 1}$$

where Ψ is the digamma function, defined as

$$\Psi(x) = \frac{\partial}{\partial x} \log \Gamma(x).$$

Sampling Statement

`n ~ ` **`neg_binomial`**`(alpha, beta);`
Increment log probability with `neg_binomial_log(n, alpha, beta)`, dropping constant additive terms; Section 23.3 explains sampling statements.

Stan Functions

real **neg_binomial_log**(ints *n*, reals *alpha*, reals *beta*)
The log negative binomial probability mass of *n* given shape *alpha* and inverse scale *beta*

real **neg_binomial_cdf**(ints *n*, reals *alpha*, reals *beta*)
The negative binomial cumulative distribution function of *n* given shape *alpha* and inverse scale *beta*

```
real neg_binomial_cdf_log(ints n, reals alpha, reals beta)
```
The log of the negative binomial cumulative distribution function of *n* given shape *alpha* and inverse scale *beta*

```
real neg_binomial_ccdf_log(ints n, reals alpha, reals beta)
```
The log of the negative binomial complementary cumulative distribution function of *n* given shape *alpha* and inverse scale *beta*

```
int neg_binomial_rng(real alpha, real beta)
```
Generate a negative binomial variate with shape *alpha* and inverse scale *beta*; may only be used in generated quantities block

34.2.   Poisson Distribution

Probability Mass Function

If $\lambda \in \mathbb{R}^+$, then for $n \in \mathbb{N}$,

$$\text{Poisson}(n|\lambda) = \frac{1}{n!} \lambda^n \exp(-\lambda).$$

Sampling Statement

```
 n ~ poisson(lambda);
```
Increment log probability with poisson_log(n,lambda), dropping constant additive terms; Section 23.3 explains sampling statements.

Stan Functions

```
real poisson_log(ints n, reals lambda)
```
The log Poisson probability mass of *n* given rate *lambda*

```
real poisson_cdf(ints n, reals lambda)
```
The Poisson cumulative distribution function of *n* given rate *lambda*

```
real poisson_cdf_log(ints n, reals lambda)
```
The log of the Poisson cumulative distribution function of *n* given rate *lambda*

```
real poisson_ccdf_log(ints n, reals lambda)
```
The log of the Poisson complementary cumulative distribution function of *n* given rate *lambda*

```
int poisson_rng(real lambda)
```
Generate a poisson variate with rate *lambda*; may only be used in generated quantities block

34.3.   Poisson Distribution, Log Parameterization

Stan also provides a parameterization of the Poisson using the log rate $\alpha = \log \lambda$ as a parameter. This is useful for log-linear Poisson regressions so that the predictor does not need to be exponentiated and passed into the standard Poisson probability function.

Probability Mass Function

If $\alpha \in \mathbb{R}$, then for $n \in \mathbb{N}$,

$$\mathsf{PoissonLog}(n|\alpha) = \frac{1}{n!} \exp\left(n\alpha - \exp(\alpha)\right).$$

Sampling Statement

```
n ~ poisson_log(alpha);
```
Increment log probability with `poisson_log_log(n,alpha)`, dropping constant additive terms; Section 23.3 explains sampling statements.

Stan Functions

```
real poisson_log_log(ints n, reals alpha)
```
The log Poisson probability mass of *n* given log rate *alpha*

35.   Multivariate Discrete Distributions

The multivariate discrete distributions are over multiple integer values, which are expressed in Stan as arrays.

35.1.   Multinomial Distribution

Probability Mass Function

If $K \in \mathbb{N}$, $N \in \mathbb{N}$, and $\theta \in K$-simplex, then for $y \in \mathbb{N}^K$ such that $\sum_{k=1}^{K} y_k = N$,

$$\text{Multinomial}(y|\theta, N) = \binom{N}{y_1, \ldots, y_K} \prod_{k=1}^{K} \theta_k^{y_k},$$

where the multinomial coefficient is defined by

$$\binom{N}{y_1, \ldots, y_k} = \frac{N!}{\prod_{k=1}^{K} y_k!}.$$

Sampling Statement

`y ~ multinomial(theta,N);`

　　Increment log probability with `multinomial_log(y,theta,N)`, dropping constant additive terms; Section 23.3 explains sampling statements.

Stan Functions

real **multinomial_log**(int[] *y*, vector *theta*, int *N*)

　　The log multinomial probability mass function with outcome array y of size K given the K-simplex distribution parameter *theta* and (implicit) total count N = `sum(y)`

vector **multinomial_rng**(vector *theta*, int *N*)

　　Generate a multinomial variate with simplex distribution parameter *theta* and (implicit) total count N; may only be used in generated quantities block

Part VII

Continuous Distributions

36. Unbounded Continuous Distributions

The unbounded univariate continuous probability distributions have support on all real numbers.

36.1. Normal Distribution

Probability Density Function

If $\mu \in \mathbb{R}$ and $\sigma \in \mathbb{R}^+$, then for $y \in \mathbb{R}$,

$$\text{Normal}(y|\mu, \sigma) = \frac{1}{\sqrt{2\pi}\,\sigma} \exp\left(-\frac{1}{2}\left(\frac{y-\mu}{\sigma}\right)^2\right).$$

Sampling Statement

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

Increment log probability with `normal_log(y,mu,sigma)`, dropping constant additive terms; Section 23.3 explains sampling statements.

Stan Functions

real **normal_log**(reals *y*, reals *mu*, reals *sigma*)

The log of the normal density of *y* given location *mu* and scale *sigma*

real **normal_cdf**(reals *y*, reals *mu*, reals *sigma*)

The cumulative normal distribution of *y* given location *mu* and scale *sigma*

real **normal_cdf_log**(reals *y*, reals *mu*, reals *sigma*)

The log of the cumulative normal distribution of *y* given location *mu* and scale *sigma*

real **normal_ccdf_log**(reals *y*, reals *mu*, reals *sigma*)

The log of the complementary cumulative normal distribution of *y* given location *mu* and scale *sigma*

real **normal_rng**(real *mu*, real *sigma*)

Generate a normal variate with location *mu* and scale *sigma*; may only be used in generated quantities block

36.2.   Exponentially Modified Normal Distribution

Probability Density Function

If $\mu \in \mathbb{R}$, $\sigma \in \mathbb{R}^+$, and $\lambda \in \mathbb{R}^+$, then for $y \in \mathbb{R}$,

$$\text{ExpModNormal}(y|\mu, \sigma, \lambda) = \frac{\lambda}{\sqrt{\pi}} \; \exp\left(\frac{\lambda}{2}\left(2\mu + \lambda\sigma^2 - 2y\right)\right) \text{erfc}\left(\frac{\mu + \lambda\sigma^2 - y}{\sqrt{2}\sigma}\right).$$

Sampling Statement

`y ~ ` **`exp_mod_normal`**`(mu,sigma,lambda);`

Increment log probability with exp_mod_normal_log(*y,mu,sigma,lambda*), dropping constant additive terms; Section 23.3 explains sampling statements.

Stan Functions

real **exp_mod_normal_log**(reals *y*, reals *mu*, reals *sigma*,
                        reals *lambda*)

The log of the exponentially modified normal density of *y* given location *mu*, scale *sigma*, and shape *lambda*

real **exp_mod_normal_cdf**(reals *y*, reals *mu*, reals *sigma*,
                        reals *lambda*)

The exponentially modified normal cumulative distribution function of *y* given location *mu*, scale *sigma*, and shape *lambda*

real **exp_mod_normal_cdf_log**(reals *y*, reals *mu*, reals *sigma*,
                        reals *lambda*)

The log of the exponentially modified normal cumulative distribution function of *y* given location *mu*, scale *sigma*, and shape *lambda*

real **exp_mod_normal_ccdf_log**(reals *y*, reals *mu*, reals *sigma*,
                        reals *lambda*)

The log of the exponentially modified normal complementary cumulative distribution function of *y* given location *mu*, scale *sigma*, and shape *lambda*

real **exp_mod_normal_rng**(real *mu*, real *sigma*, real *lambda*)

Generate a exponentially modified normal variate with location *mu*, scale *sigma*, and shape *lambda*; may only be used in generated quantities block

36.3. Skew Normal Distribution

Probabilty Density Function

If $\mu \in \mathbb{R}$, $\sigma \in \mathbb{R}^+$, and $\alpha \in \mathbb{R}$, then for $y \in \mathbb{R}$,

$$\text{SkewNormal}(y|\mu,\sigma,\alpha) = \frac{1}{\sigma\sqrt{2\pi}} \; \exp\left(-\frac{1}{2}\left(\frac{y-\mu}{\sigma}\right)^2\right) \left(1 + \text{erf}\left(\alpha\left(\frac{y-\mu}{\sigma\sqrt{2}}\right)\right)\right).$$

Sampling Statement

`y ~ ` **`skew_normal`**`(mu,sigma,alpha);`
Increment log probability with `skew_normal_log(y,mu,sigma,alpha)`, dropping constant additive terms; Section 23.3 explains sampling statements.

Stan Functions

real **skew_normal_log**(reals *y*, reals *mu*, reals *sigma*, reals *alpha*)
The log of the skew normal density of *y* given location *mu*, scale *sigma*, and shape *alpha*

real **skew_normal_cdf**(reals *y*, reals *mu*, reals *sigma*, reals *alpha*)
The skew normal distribution function of *y* given location *mu*, scale *sigma*, and shape *alpha*

real **skew_normal_cdf_log**(reals *y*, reals *mu*, reals *sigma*, reals *alpha*)
The log of the skew normal cumulative distribution function of *y* given location *mu*, scale *sigma*, and shape *alpha*

real **skew_normal_ccdf_log**(reals *y*, reals *mu*, reals *sigma*, reals *alpha*)
The log of the skew normal complementary cumulative distribution function of *y* given location *mu*, scale *sigma*, and shape *alpha*

real **skew_normal_rng**(real *mu*, real *sigma*, real *alpha*)
Generate a skew normal variate with location *mu*, scale *sigma*, and shape *alpha*; may only be used in generated quantities block

36.4. Student-*t* Distribution

Probability Density Function

If $\nu \in \mathbb{R}^+$, $\mu \in \mathbb{R}$, and $\sigma \in \mathbb{R}^+$, then for $y \in \mathbb{R}$,

$$\mathsf{StudentT}(y|\nu, \mu, \sigma) = \frac{\Gamma\left((\nu + 1)/2\right)}{\Gamma(\nu/2)} \; \frac{1}{\sqrt{\nu\pi}\,\sigma} \; \left(1 + \frac{1}{\nu}\left(\frac{y - \mu}{\sigma}\right)^2\right)^{-(\nu+1)/2}.$$

Sampling Statement

`y ~ `**`student_t`**`(nu,mu,sigma);`

Increment log probability with `student_t_log(y,nu,mu,sigma)`, dropping constant additive terms; Section 23.3 explains sampling statements.

Stan Functions

real **student_t_log**(reals *y*, reals *nu*, reals *mu*, reals *sigma*)

The log of the Student-*t* density of *y* given degrees of freedom *nu*, location *mu*, and scale *sigma*

real **student_t_cdf**(reals *y*, reals *nu*, reals *mu*, reals *sigma*)

The Student-*t* cumulative distribution function of *y* given degrees of freedom *nu*, location *mu*, and scale *sigma*

real **student_t_cdf_log**(reals *y*, reals *nu*, reals *mu*, reals *sigma*)

The log of the Student-*t* cumulative distribution function of *y* given degrees of freedom *nu*, location *mu*, and scale *sigma*

real **student_t_ccdf_log**(reals *y*, reals *nu*, reals *mu*, reals *sigma*)

The log of the Student-*t* complementary cumulative distribution function of *y* given degrees of freedom *nu*, location *mu*, and scale *sigma*

real **student_t_rng**(real *nu*, real *mu*, real *sigma*)

Generate a Student-*t* variate with degrees of freedom *nu*, location *mu*, and scale *sigma*; may only be used in generated quantities block

36.5. Cauchy Distribution

Probability Density Function

If $\mu \in \mathbb{R}$ and $\sigma \in \mathbb{R}^+$, then for $y \in \mathbb{R}$,

$$\text{Cauchy}(y|\mu, \sigma) = \frac{1}{\pi\sigma} \, \frac{1}{1 + ((y - \mu)/\sigma)^2}.$$

Sampling Statement

`y ~` **`cauchy`**`(`*`mu,sigma`*`);`

Increment log probability with `cauchy_log(`*`y,mu,sigma`*`)`, dropping constant additive terms; Section 23.3 explains sampling statements.

Stan Functions

`real` **`cauchy_log`**`(reals y, reals `*`mu`*`, reals `*`sigma`*`)`

The log of the Cauchy density of *y* given location *mu* and scale *sigma*

`real` **`cauchy_cdf`**`(reals y, reals `*`mu`*`, reals `*`sigma`*`)`

The Cauchy cumulative distribution function of *y* given location *mu* and scale *sigma*

`real` **`cauchy_cdf_log`**`(reals y, reals `*`mu`*`, reals `*`sigma`*`)`

The log of the Cauchy cumulative distribution function of *y* given location *mu* and scale *sigma*

`real` **`cauchy_ccdf_log`**`(reals y, reals `*`mu`*`, reals `*`sigma`*`)`

The log of the Cauchy complementary cumulative distribution function of *y* given location *mu* and scale *sigma*

`real` **`cauchy_rng`**`(real `*`mu`*`, real `*`sigma`*`)`

Generate a Cauchy variate with location *mu* and scale *sigma*; may only be used in generated quantities block

36.6. Double Exponential (Laplace) Distribution

Probability Density Function

If $\mu \in \mathbb{R}$ and $\sigma \in \mathbb{R}^+$, then for $y \in \mathbb{R}$,

$$\text{DoubleExponential}(y|\mu, \sigma) = \frac{1}{2\sigma} \exp\left(-\frac{|y - \mu|}{\sigma}\right).$$

`y ~` **`double_exponential`**`(`*`mu`*`,`*`sigma`*`);`

Increment log probability with `double_exponential_log(`*`y`*`,`*`mu`*`,`*`sigma`*`)`, dropping constant additive terms; Section 23.3 explains sampling statements.

Stan Functions

`real` **`double_exponential_log`**`(reals `*`y`*`, reals `*`mu`*`, reals `*`sigma`*`)`

The log of the double exponential density of *y* given location *mu* and scale *sigma*

`real` **`double_exponential_cdf`**`(reals `*`y`*`, reals `*`mu`*`, reals `*`sigma`*`)`

The double exponential cumulative distribution function of *y* given location *mu* and scale *sigma*

`real` **`double_exponential_cdf_log`**`(reals `*`y`*`, reals `*`mu`*`, reals `*`sigma`*`)`

The log of the double exponential cumulative distribution function of *y* given location *mu* and scale *sigma*

`real` **`double_exponential_ccdf_log`**`(reals `*`y`*`, reals `*`mu`*`, reals `*`sigma`*`)`

The log of the double exponential complementary cumulative distribution function of *y* given location *mu* and scale *sigma*

`real` **`double_exponential_rng`**`(real `*`mu`*`, real `*`sigma`*`)`

Generate a double exponential variate with location *mu* and scale *sigma*; may only be used in generated quantities block

36.7. Logistic Distribution

Probability Density Function

If $\mu \in \mathbb{R}$ and $\sigma \in \mathbb{R}^+$, then for $y \in \mathbb{R}$,

$$\text{Logistic}(y|\mu,\sigma) = \frac{1}{\sigma} \, \exp\left(-\frac{y-\mu}{\sigma}\right) \, \left(1 + \exp\left(-\frac{y-\mu}{\sigma}\right)\right)^{-2}.$$

Sampling Statement

`y ~` **`logistic`**`(`*`mu`*`,`*`sigma`*`);`

Increment log probability with `logistic_log(`*`y`*`,`*`mu`*`,`*`sigma`*`)`, dropping constant additive terms; Section 23.3 explains sampling statements.

real **logistic_log**(reals *y*, reals *mu*, reals *sigma*)

    The log of the logistic density of *y* given location *mu* and scale *sigma*

real **logistic_cdf**(reals *y*, reals *mu*, reals *sigma*)

    The logistic cumulative distribution function of *y* given location *mu* and scale *sigma*

real **logistic_cdf_log**(reals *y*, reals *mu*, reals *sigma*)

    The log of the logistic cumulative distribution function of *y* given location *mu* and scale *sigma*

real **logistic_ccdf_log**(reals *y*, reals *mu*, reals *sigma*)

    The log of the logistic complementary cumulative distribution function of *y* given location *mu* and scale *sigma*

real **logistic_rng**(real *mu*, real *sigma*)

    Generate a logistic variate with location *mu* and scale *sigma*; may only be used in generated quantities block

36.8. Gumbel Distribution

Probability Density Function

If $\mu \in \mathbb{R}$ and $\beta \in \mathbb{R}^+$, then for $y \in \mathbb{R}$,

$$\text{Gumbel}(y|\mu, \beta) = \frac{1}{\beta} \, \exp\left(-\frac{y-\mu}{\beta} - \exp\left(-\frac{y-\mu}{\beta}\right)\right).$$

Sampling Statement

  y ~ **gumbel**(*mu,beta*);

    Increment log probability with gumbel_log(*y,mu,beta*), dropping constant additive terms; Section 23.3 explains sampling statements.

Stan Functions

real **gumbel_log**(reals *y*, reals *mu*, reals *beta*)

    The log of the gumbel density of *y* given location *mu* and scale *beta*

real **gumbel_cdf**(reals *y*, reals *mu*, reals *beta*)

> The gumbel cumulative distribution function of *y* given location *mu* and scale *beta*

real **gumbel_cdf_log**(reals *y*, reals *mu*, reals *beta*)

> The log of the gumbel cumulative distribution function of *y* given location *mu* and scale *beta*

real **gumbel_ccdf_log**(reals *y*, reals *mu*, reals *beta*)

> The log of the gumbel complementary cumulative distribution function of *y* given location *mu* and scale *beta*

real **gumbel_rng**(real *mu*, real *beta*)

> Generate a gumbel variate with location *mu* and scale *beta*; may only be used in generated quantities block

37. Positive Continuous Distributions

The positive continuous probability functions have support on the positive real numbers.

37.1. Lognormal Distribution

Probability Density Function

If $\mu \in \mathbb{R}$ and $\sigma \in \mathbb{R}^+$, then for $y \in \mathbb{R}^+$,

$$\mathsf{LogNormal}(y|\mu,\sigma) = \frac{1}{\sqrt{2\pi}\,\sigma}\,\frac{1}{y}\,\exp\left(-\frac{1}{2}\left(\frac{\log y - \mu}{\sigma}\right)^2\right).$$

Sampling Statement

`y ~ `**`lognormal`**`(`*`mu,sigma`*`);`
Increment log probability with `lognormal_log(`*`y,mu,sigma`*`)`, dropping constant additive terms; Section 23.3 explains sampling statements.

Stan Functions

real **`lognormal_log`**`(reals `*`y`*`, reals `*`mu`*`, reals `*`sigma`*`)`
The log of the lognormal density of *y* given location *mu* and scale *sigma*

real **`lognormal_cdf`**`(reals `*`y`*`, reals `*`mu`*`, reals `*`sigma`*`)`
The cumulative lognormal distribution function of *y* given location *mu* and scale *sigma*

real **`lognormal_cdf_log`**`(reals `*`y`*`, reals `*`mu`*`, reals `*`sigma`*`)`
The log of the lognormal cumulative distribution fucntion of *y* given location *mu* and scale *sigma*

real **`lognormal_ccdf_log`**`(reals `*`y`*`, reals `*`mu`*`, reals `*`sigma`*`)`
The log of the lognormal complementary cumulative distribution function of *y* given location *mu* and scale *sigma*

real **`lognormal_rng`**`(real `*`mu`*`, real `*`beta`*`)`
Generate a lognormal variate with location *mu* and scale *sigma*; may only be used in generated quantities block

37.2. Chi-Square Distribution

Probability Density Function

If $\nu \in \mathbb{R}^+$, then for $y \in \mathbb{R}^+$,

$$\text{ChiSquare}(y|\nu) = \frac{2^{-\nu/2}}{\Gamma(\nu/2)} \, y^{\nu/2-1} \, \exp\left(-\frac{1}{2}\,y\right).$$

Sampling Statement

`y ~` **`chi_square`**`(`*`nu`*`);`

Increment log probability with `chi_square_log(`*`y`*`,`*`nu`*`)`, dropping constant additive terms; Section 23.3 explains sampling statements.

Stan Functions

real **`chi_square_log`**`(reals `*`y`*`, reals `*`nu`*`)`

The log of the Chi-square density of *y* given degrees of freedom *nu*

real **`chi_square_cdf`**`(reals `*`y`*`, reals `*`nu`*`)`

The Chi-square cumulative distribution function of *y* given degrees of freedom *nu*

real **`chi_square_cdf_log`**`(reals `*`y`*`, reals `*`nu`*`)`

The log of the Chi-square cumulative distribution function of *y* given degrees of freedom *nu*

real **`chi_square_ccdf_log`**`(reals `*`y`*`, reals `*`nu`*`)`

The log of the complementary Chi-square cumulative distribution function of *y* given degrees of freedom *nu*

real **`chi_square_rng`**`(real `*`nu`*`)`

Generate a Chi-square variate with degrees of freedom *nu*; may only be used in generated quantities block

37.3. Inverse Chi-Square Distribution

Probability Density Function

If $\nu \in \mathbb{R}^+$, then for $y \in \mathbb{R}^+$,

$$\text{InvChiSquare}(y|\nu) = \frac{2^{-\nu/2}}{\Gamma(\nu/2)} \, y^{-(\nu/2-1)} \, \exp\left(-\frac{1}{2}\,\frac{1}{y}\right).$$

`y ~` **`inv_chi_square`**`(`*`nu`*`);`
Increment log probability with `inv_chi_square_log(`*y*`,`*nu*`)`, dropping constant additive terms; Section 23.3 explains sampling statements.

Stan Functions

`real` **`inv_chi_square_log`**`(reals` *y*`, reals` *nu*`)`
The log of the inverse Chi-square density of *y* given degrees of freedom *nu*

`real` **`inv_chi_square_cdf`**`(reals` *y*`, reals` *nu*`)`
The inverse Chi-squared cumulative distribution function of *y* given degrees of freedom *nu*

`real` **`inv_chi_square_cdf_log`**`(reals` *y*`, reals` *nu*`)`
The log of the inverse Chi-squared cumulative distribution function of *y* given degrees of freedom *nu*

`real` **`inv_chi_square_ccdf_log`**`(reals` *y*`, reals` *nu*`)`
The log of the inverse Chi-squared complementary cumulative distribution function of *y* given degrees of freedom *nu*

`real` **`inv_chi_square_rng`**`(real` *nu*`)`
Generate an inverse Chi-squared variate with degrees of freedom *nu*; may only be used in generated quantities block

37.4. Scaled Inverse Chi-Square Distribution

Probability Density Function

If $\nu \in \mathbb{R}^+$ and $\sigma \in \mathbb{R}^+$, then for $y \in \mathbb{R}^+$,

$$\text{ScaledInvChiSquare}(y|\nu, \sigma) = \frac{(\nu/2)^{\nu/2}}{\Gamma(\nu/2)} \, \sigma^\nu \, y^{-(\nu/2+1)} \, \exp\left(-\frac{1}{2} \, \nu \, \sigma^2 \, \frac{1}{y}\right).$$

Sampling Statement

`y ~` **`scaled_inv_chi_square`**`(`*`nu`*`,`*`sigma`*`);`
Increment log probability with `scaled_inv_chi_square_log(`*y*`,`*nu*`,`*sigma*`)`, dropping constant additive terms; Section 23.3 explains sampling statements.

real **scaled_inv_chi_square_log**(reals *y*, reals *nu*, reals *sigma*)

 The log of the scaled inverse Chi-square density of *y* given degrees of freedom *nu* and scale *sigma*

real **scaled_inv_chi_square_cdf**(reals *y*, reals *nu*, reals *sigma*)

 The scaled inverse Chi-square cumulative distribution function of *y* given degrees of freedom *nu* and scale *sigma*

real **scaled_inv_chi_square_cdf_log**(reals *y*, reals *nu*, reals *sigma*)

 The log of the scaled inverse Chi-square cumulative distribution function of *y* given degrees of freedom *nu* and scale *sigma*

real **scaled_inv_chi_square_ccdf_log**(reals *y*, reals *nu*, reals *sigma*)

 The log of the scaled inverse Chi-square complementary cumulative distribution function of *y* given degrees of freedom *nu* and scale *sigma*

real **scaled_inv_chi_square_rng**(real *nu*, real *sigma*)

 Generate a scaled inverse Chi-squared variate with degrees of freedom *nu* and scale *sigma*; may only be used in generated quantities block

37.5. Exponential Distribution

Probability Density Function

If $\beta \in \mathbb{R}^+$, then for $y \in \mathbb{R}^+$,

$$\text{Exponential}(y|\beta) = \beta \exp(-\beta y).$$

Sampling Statement

 y ~ **exponential**(*beta*);

 Increment log probability with exponential_log(*y*,*beta*), dropping constant additive terms; Section 23.3 explains sampling statements.

Stan Functions

real **exponential_log**(reals *y*, reals *beta*)

 The log of the exponential density of *y* given inverse scale *beta*

real **exponential_cdf**(reals *y*, reals *beta*)

    The exponential cumulative distribution function of *y* given inverse scale *beta*

real **exponential_cdf_log**(reals *y*, reals *beta*)

    The log of the exponential cumulative distribution function of *y* given inverse scale *beta*

real **exponential_ccdf_log**(reals *y*, reals *beta*)

    The log of the exponential complementary cumulative distribution function of *y* given inverse scale *beta*

real **exponential_rng**(real *beta*)

    Generate an exponential variate with inverse scale *beta*; may only be used in generated quantities block

37.6. Gamma Distribution

Probability Density Function

If $\alpha \in \mathbb{R}^+$ and $\beta \in \mathbb{R}^+$, then for $y \in \mathbb{R}^+$,

$$\mathsf{Gamma}(y|\alpha, \beta) = \frac{\beta^\alpha}{\Gamma(\alpha)} \, y^{\alpha-1} \exp(-\beta \, y).$$

Sampling Statement

 y ~ **gamma**(*alpha,beta*);

    Increment log probability with gamma_log(*y,alpha,beta*), dropping constant additive terms; Section 23.3 explains sampling statements.

Stan Functions

real **gamma_log**(reals *y*, reals *alpha*, reals *beta*)

    The log of the gamma density of *y* given shape *alpha* and inverse scale *beta*

real **gamma_cdf**(reals *y*, reals *alpha*, reals *beta*)

    The cumulative gamma distribution function of *y* given shape *alpha* and inverse scale *beta*

real **gamma_cdf_log**(reals *y*, reals *alpha*, reals *beta*)

    The log of the cumulative gamma distribution function of *y* given shape *alpha* and inverse scale *beta*

real **gamma_ccdf_log**(reals *y*, reals *alpha*, reals *beta*)

> The log of the complementary cumulative gamma distribution function of *y* given shape *alpha* and inverse scale *beta*

real **gamma_rng**(real *alpha*, real *beta*)

> Generate a gamma variate with shape *alpha* and inverse scale *beta*; may only be used in generated quantities block

37.7. Inverse Gamma Distribution

Probability Density Function

If $\alpha \in \mathbb{R}^+$ and $\beta \in \mathbb{R}^+$, then for $y \in \mathbb{R}^+$,

$$\mathsf{InvGamma}(y|\alpha, \beta) = \frac{\beta^\alpha}{\Gamma(\alpha)} \; y^{-(\alpha+1)} \; \exp\!\left(-\beta \, \frac{1}{y}\right).$$

Sampling Statement

`y ~ `**`inv_gamma`**`(alpha,beta);`

> Increment log probability with `inv_gamma_log(y,alpha,beta)`, dropping constant additive terms; Section 23.3 explains sampling statements.

Stan Functions

real **inv_gamma_log**(reals *y*, reals *alpha*, reals *beta*)

> The log of the inverse gamma density of *y* given shape *alpha* and scale *beta*

real **inv_gamma_cdf**(reals *y*, reals *alpha*, reals *beta*)

> The inverse gamma cumulative distribution function of *y* given shape *alpha* and scale *beta*

real **inv_gamma_cdf_log**(reals *y*, reals *alpha*, reals *beta*)

> The log of the inverse gamma cumulative distribution function of *y* given shape *alpha* and scale *beta*

real **inv_gamma_ccdf_log**(reals *y*, reals *alpha*, reals *beta*)

> The log of the inverse gamma complementary cumulative distribution function of *y* given shape *alpha* and scale *beta*

real **inv_gamma_rng**(real *alpha*, real *beta*)

> Generate an inverse gamma variate with shape *alpha* and scale *beta*; may only be used in generated quantities block

37.8. Weibull Distribution

Probability Density Function

If $\alpha \in \mathbb{R}^+$ and $\sigma \in [0, \infty)$, then for $y \in \mathbb{R}^+$,

$$\text{Weibull}(y|\alpha, \sigma) = \frac{\alpha}{\sigma} \left(\frac{y}{\sigma}\right)^{\alpha-1} \exp\left(-\left(\frac{y}{\sigma}\right)^{\alpha}\right).$$

Sampling Statement

`y ~ `**`weibull`**`(`*`alpha,sigma`*`);`

Increment log probability with `weibull_log(`*`y,alpha,sigma`*`)`, dropping constant additive terms; Section 23.3 explains sampling statements.

Stan Functions

real **weibull_log**(reals *y*, reals *alpha*, reals *sigma*)

The log of the Weibull density of *y* given shape *alpha* and scale *sigma*

real **weibull_cdf**(reals *y*, reals *alpha*, reals *sigma*)

The Weibull cumulative distribution function of *y* given shape *alpha* and scale *sigma*

real **weibull_cdf_log**(reals *y*, reals *alpha*, reals *sigma*)

The log of the Weibull cumulative distribution function of *y* given shape *alpha* and scale *sigma*

real **weibull_ccdf_log**(reals *y*, reals *alpha*, reals *sigma*)

The log of the Weibull complementary cumulative distribution function of *y* given shape *alpha* and scale *sigma*

real **weibull_rng**(real *alpha*, real *sigma*)

Generate a weibull variate with shape *alpha* and scale *sigma*; may only be used in generated quantities block

38. Non-negative Continuous Distributions

The non-negative continuous probability functions have support on the non-negative real numbers.

38.1. Rayleigh Distribution

Probability Density Function

If $\sigma \in \mathbb{R}^+$, then for $y \in [0, \infty)$,

$$\text{Rayleigh}(y|\sigma) = \frac{y}{\sigma^2} \exp(-y^2/2\sigma^2).$$

Sampling Statement

`y ~ `**`rayleigh`**`(`*`sigma`*`);`
   Increment log probability with `rayleigh_log(`*`y,sigma`*`)`, dropping constant additive terms; Section 23.3 explains sampling statements.

Stan Functions

real **rayleigh_log**(reals *y*, reals *sigma*)
   The log of the Rayleigh ensity of *y* given scale *sigma*

real **rayleigh_cdf**(real *y*, real *sigma*)
   The Rayleigh cumulative distribution of *y* given scale *sigma*

real **rayleigh_cdf_log**(real *y*, real *sigma*)
   The log of the Rayleigh cumulative distribution of *y* given scale *sigma*

real **rayleigh_ccdf_log**(real *y*, real *sigma*)
   The log of the Rayleigh complementary cumulative distribution of *y* given scale *sigma*

real **rayleigh_rng**(real *sigma*)
   Generate a Rayleigh variate with scale *sigma*; may only be used in generated quantities block

39. Positive Lower-Bounded Probabilities

The positive lower-bounded probabilities have support on real values above some positive minimum value.

39.1. Pareto Distribution

Probability Density Function

If $y_0 \in \mathbb{R}^+$ and $\alpha \in \mathbb{R}^+$, then for $y \in \mathbb{R}^+$,

$$\text{Pareto}(y|y_0, \alpha) = \alpha \; y_0 \; \left(\frac{1}{y}\right)^{\alpha+1}.$$

Sampling Statement

`y ~ pareto(y_min,alpha);`

Increment log probability with `pareto_log(y,y_min,alpha)`, dropping constant additive terms; Section 23.3 explains sampling statements.

Stan Functions

real **pareto_log**(reals *y*, reals *y_min*, reals *alpha*)

The log of the Pareto density of *y* given positive minimum value *y_min* and shape *alpha*

real **pareto_cdf**(reals *y*, reals *y_min*, reals *alpha*)

The Pareto cumulative distribution function of *y* given positive minimum value *y_min* and shape *alpha*

real **pareto_cdf_log**(reals *y*, reals *y_min*, reals *alpha*)

The log of the Pareto cumulative distribution function of *y* given positive minimum value *y_min* and shape *alpha*

real **pareto_ccdf_log**(reals *y*, reals *y_min*, reals *alpha*)

The log of the Pareto complementary cumulative distribution function of *y* given positive minimum value *y_min* and shape *alpha*

real **pareto_rng**(real *y_min*, real *alpha*)

Generate a Pareto variate with positive minimum value *y_min* and shape *alpha*; may only be used in generated quantities block

40. Continuous Distributions on [0, 1]

The continuous distributions with outcomes in the interval $[0, 1]$ are used to characterized bounded quantities, including probabilities.

40.1. Beta Distribution

Probability Density Function

If $\alpha \in \mathbb{R}^+$ and $\beta \in \mathbb{R}^+$, then for $\theta \in (0, 1)$,

$$\text{Beta}(\theta|\alpha, \beta) = \frac{1}{\text{B}(\alpha, \beta)} \, \theta^{\alpha-1} \, (1-\theta)^{\beta-1},$$

where the beta function B() is as defined in Section 29.11. If $\theta = 0$ or $\theta = 1$, then the log probability is $-\infty$.

Sampling Statement

`theta ~ `**`beta`**`(alpha,beta);`

Increment log probability with `beta_log(`*`theta,alpha,beta`*`)`, dropping constant additive terms; Section 23.3 explains sampling statements.

Stan Functions

real **beta_log**(reals *theta*, reals *alpha*, reals *beta*)

The log of the beta density of `theta` in $[0, 1]$ given positive prior successes (plus one) *alpha* and prior failures (plus one) *beta*

real **beta_cdf**(reals *theta*, reals *alpha*, reals *beta*)

The beta cumulative distribution function of `theta` in $[0, 1]$ given positive prior successes (plus one) *alpha* and prior failures (plus one) *beta*

real **beta_cdf_log**(reals *theta*, reals *alpha*, reals *beta*)

The log of the beta cumulative distribution function of `theta` in $[0, 1]$ given positive prior successes (plus one) *alpha* and prior failures (plus one) *beta*

real **beta_ccdf_log**(reals *theta*, reals *alpha*, reals *beta*)

The log of the beta complementary cumulative distribution function of `theta` in $[0, 1]$ given positive prior successes (plus one) *alpha* and prior failures (plus one) *beta*

`real` **`beta_rng`**`(real `*`alpha`*`, real `*`beta`*`)`

Generate a beta variate with positive prior successes (plus one) *alpha* and prior failures (plus one) *beta*; may only be used in generated quantities block

41.   Circular Distributions

Circular distributions are defined for finite values y in any interval of length 2π.

41.1.   Von Mises Distribution

Probability Density Function

If $\mu \in \mathbb{R}$ and $\kappa \in \mathbb{R}^+$, then for $y \in \mathbb{R}$,

$$\text{VonMises}(y|\mu, \kappa) = \frac{\exp(\kappa \cos(y - \mu))}{2\pi I_0(\kappa)}.$$

Sampling Statement

`y ~` **`von_mises`**`(mu,kappa);`
   Increment log probability with `von_mises_log(`*`y`*`,`*`mu`*`,`*`kappa`*`)`, dropping constant additive terms; Section 23.3 explains sampling statements.

Stan Functions

`real` **`von_mises_log`**`(reals y, reals mu, reals kappa)`
   The log of the von mises density of *y* given location *mu* and scale *kappa*

42. Bounded Continuous Probabilities

The bounded continuous probabilities have support on a finite interval of real numbers.

42.1. Uniform Distribution

Probability Density Function

If $\alpha \in \mathbb{R}$ and $\beta \in (\alpha, \infty)$, then for $y \in [\alpha, \beta]$,

$$\text{Uniform}(y|\alpha, \beta) = \frac{1}{\beta - \alpha}.$$

Sampling Statement

`y ~ `**`uniform`**`(alpha,beta);`

Increment log probability with `uniform_log(y,alpha,beta)`, dropping constant additive terms; Section 23.3 explains sampling statements.

Stan Functions

real **uniform_log**(reals *y*, reals *alpha*, reals *beta*)

The log of the uniform density of *y* given lower bound *alpha* and upper bound *beta*

real **uniform_cdf**(reals *y*, reals *alpha*, reals *beta*)

The uniform cumulative distribution function of *y* given lower bound *alpha* and upper bound *beta*

real **uniform_cdf_log**(reals *y*, reals *alpha*, reals *beta*)

The log of the uniform cumulative distribution function of *y* given lower bound *alpha* and upper bound *beta*

real **uniform_ccdf_log**(reals *y*, reals *alpha*, reals *beta*)

The log of the uniform complementary cumulative distribution function of *y* given lower bound *alpha* and upper bound *beta*

real **uniform_rng**(real *alpha*, real *beta*)

Generate a uniform variate with lower bound *alpha* and upper bound *beta*; may only be used in generated quantities block

43. Distributions over Unbounded Vectors

The unbounded vector probability distributions have support on all of $\mathbb{R}^K$ for some fixed K.

43.1. Multivariate Normal Distribution

Probability Density Function

If $K \in \mathbb{N}$, $\mu \in \mathbb{R}^K$, and $\Sigma \in \mathbb{R}^{K \times K}$ is symmetric and positive definite, then for $y \in \mathbb{R}^K$,

$$\mathsf{MultiNormal}(y|\mu, \Sigma) = \frac{1}{(2\pi)^{K/2}} \; \frac{1}{\sqrt{|\Sigma|}} \; \exp\left(-\frac{1}{2}(y - \mu)^\top \Sigma^{-1} (y - \mu)\right),$$

where $|\Sigma|$ is the absolute determinant of Σ.

Sampling Statement

```
y ~ multi_normal(mu,Sigma);
```
Increment log probability with `multi_normal_log(y,mu,Sigma)`, dropping constant additive terms; Section 23.3 explains sampling statements.

Stan Functions

real **multi_normal_log**(vector *y*, vector *mu*, matrix *Sigma*)

The log of the multivariate normal density of vector *y* given location *mu* and covariance matrix *Sigma*

vector **multi_normal_rng**(vector *mu*, matrix *Sigma*)

Generate a multivariate normal variate with location *mu* and covariance matrix *Sigma*; may only be used in generated quantities block

43.2. Multivariate Normal Distribution, Precision Parameterization

Probability Density Function

If $K \in \mathbb{N}$, $\mu \in \mathbb{R}^K$, and $\Omega \in \mathbb{R}^{K \times K}$ is symmetric and positive definite, then for $y \in \mathbb{R}^K$,

$$\mathsf{MultiNormalPrecision}(y|\mu, \Omega) = \mathsf{MultiNormal}(y|\mu, \Sigma^{-1})$$

 y ~ **multi_norm_prec**(*mu,Omega*);
    Increment log probability with multi_norm_prec_log(*y,mu,Omega*), dropping constant additive terms; Section 23.3 explains sampling statements.

Stan Functions

real **multi_normal_prec_log**(vector *y*, vector *mu*, matrix *Omega*)
    The log of the multivariate normal density of vector *y* given location *mu* and positive definite precision matrix *Omega*

43.3.   Multivariate Normal Distribution, Cholesky Parameterization

Probability Density Function

If $K \in \mathbb{N}$, $\mu \in \mathbb{R}^K$, and $L \in \mathbb{R}^{K \times K}$ is lower triangular and such that $LL^\top$ is positive definite, then for $y \in \mathbb{R}^K$,

$$\mathsf{MultiNormalCholesky}(y|\mu, L) = \mathsf{MultiNormal}(y|\mu, LL^\top).$$

Sampling Statement

 y ~ **multi_normal_cholesky**(*mu,L*);
    Increment log probability with multi_normal_cholesky_log(*y,mu,L*), dropping constant additive terms; Section 23.3 explains sampling statements.

Stan Functions

real **multi_normal_cholesky_log**(vector *y*, vector *mu*, matrix *L*)
    The log of the multivariate normal density of vector *y* given location *mu* and lower-triangular Cholesky factor of the covariance matrix *L*

43.4. Multivariate Student-*t* Distribution

Probability Density Function

If $K \in \mathbb{N}$, $\nu \in \mathbb{R}^+$, $\mu \in \mathbb{R}^K$, and $\Sigma \in \mathbb{R}^{K \times K}$ is symmetric and positive definite, then for $y \in \mathbb{R}^K$,

$$\text{MultiStudentT}(y|\nu, \mu, \Sigma)$$

$$= \frac{1}{\pi^{K/2}} \frac{1}{\nu^{K/2}} \frac{\Gamma((\nu + K)/2)}{\Gamma(\nu/2)} \frac{1}{\sqrt{|\Sigma|}} \left(1 + \frac{1}{\nu} (y - \mu)^\top \Sigma^{-1} (y - \mu) \right)^{-(\nu+K)/2}.$$

Sampling Statement

```
y ~ multi_student_t(nu,mu,Sigma);
```
Increment log probability with `multi_student_t_log(y,nu,mu,Sigma)`, dropping constant additive terms; Section 23.3 explains sampling statements.

Stan Functions

real **multi_student_t_log**(vector *y*, real *nu*, vector *mu*, matrix *Sigma*)

The log of the multivariate Student-*t* density of vector *y* given degrees of freedom *nu*, location *mu*, and scale matrix *Sigma*

vector **multi_student_t_rng**(real *nu*, vector *mu*, matrix *Sigma*)

Generate a multivariate Student-*t* variate with degrees of freedom *nu*, location *mu*, and scale matrix *Sigma*; may only be used in generated quantities block

43.5. Gaussian Dynamic Linear Models

A Gaussian Dynamic Linear model is defined as follows, For $t \in 1, \dots, T$,

$$y_t \sim N(F' \theta_t, V)$$
$$\theta_t \sim N(G \theta_{t-1}, W)$$
$$\theta_0 \sim N(m_0, C_0)$$

where y is $n \times T$ matrix where rows are variables and columns are observations. These functions calculate the log-likelihood of the observations marginalizing over the latent states ($p(y|F, G, V, W, m_0, C_0)$). This log-likelihood is system is calculated using

the the Kalman Filter. If *V* is diagonal, then a more efficient algorithm which sequentially processes observations and avoids a matrix inversions can be used (Durbin and Koopman, 2001, Sec 6.4).

Sampling Statement

` y ~ `**`gaussian_dlm_obs`**`(F,G,V,W,m0,C0);`
> Increment log probability with `gaussian_dlm_obs_log(y,F,G,V,W,m0,C0)`, dropping constant additive terms; Section 23.3 explains sampling statements.

Stan Functions

The following two functions differ in the type of their *V*, the first taking a full observation covariance matrix *V* and the second a vector *V* representing the diagonal of the observation covariance matrix. The sampling statement defined in the previous section works with either type of observation *V*.

`real `**`gaussian_dlm_obs_log`**`(vector y, matrix F, matrix G, matrix V, matrix W, vector m0, matrix C0)`
> The log of the density of the Gaussian Dynamic Linear model with observation matrix *y* in which rows are variables and columns are observations, design matrix *F*, transition matrix *G*, observation covariance matrix *V*, system covariance matrix *W*, and the initial state is distributed normal with mean *m0* and covariance *C0*.

`real `**`gaussian_dlm_obs_log`**`(vector y, matrix F, matrix G, vector V, matrix W, vector m0, matrix C0)`
> The log of the density of the Gaussian Dynamic Linear model with observation matrix *y* in which rows are variables and columns are observations, design matrix *F*, transition matrix *G*, observation covariance matrix with diagonal *V*, system covariance matrix *W*, and the initial state is distributed normal with mean *m0* and covariance *C0*.

44.   Simplex Distributions

The simplex probabilities have support on the unit K-simplex for a specified K. A K-dimensional vector θ is a unit K-simplex if $\theta_k \geq 0$ for $k \in \{1,\ldots,K\}$ and $\sum_{k=1}^{K} \theta_k = 1$.

44.1.   Dirichlet Distribution

Probability Density Function

If $K \in \mathbb{N}$ and $\alpha \in (\mathbb{R}^+)^K$, then for $\theta \in K$-simplex,

$$\mathsf{Dirichlet}(\theta|\alpha) = \frac{\Gamma\left(\sum_{k=1}^{K} \alpha_k\right)}{\prod_{k=1}^{K} \Gamma(\alpha_k)} \prod_{k=1}^{K} \theta_k^{\alpha_k - 1}.$$

Sampling Statement

`theta ~ `**`dirichlet`**`(alpha);`

Increment log probability with `dirichlet_log(theta,alpha)`, dropping constant additive terms; Section 23.3 explains sampling statements.

Stan Functions

`real `**`dirichlet_log`**`(vector theta, vector alpha)`

The log of the Dirichlet density for simplex *theta* given prior counts (plus one) *alpha*

`vector `**`dirichlet_rng`**`(vector alpha)`

Generate a Dirichlet variate with prior counts (plus one) *alpha*; may only be used in generated quantities block

45.   Correlation Matrix Distributions

The correlation matrix distributions have support on the (Cholesky factors of) correlation matrices. A Cholesky factor L for a $K \times K$ covariance matrix of dimensions K has rows of unit length so that the diagonal of $LL^\top$ is the unit K-vector.

45.1.   LKJ Correlation Distribution

Probability Density Function

For $\eta > 0$, if Σ a positive-definite, symmetric matrix with unit diagonal (i.e., a correlation matrix), then

$$\mathsf{LkjCorr}(\Sigma|\eta) \propto \det(\Sigma)^{(\eta-1)}.$$

The shape parameter η can be interpreted like the shape parameter of a symmetric beta distribution.

- · if $\eta = 1$, then the density is uniform over all correlation matrices of a given order;

- · if $\eta > 1$, the identity matrix is the modal correlation matrix, with sharper peaks in the density around the identity matrix for larger η; and

- · for $0 < \eta < 1$, the density has a trough at the identity matrix.

See (Lewandowski et al., 2009) for definitions.

Sampling Statement

`y ~ lkj_corr_log(eta);`
Increment log probability with `lkj_corr_log_log(y,eta)`, dropping constant additive terms; Section 23.3 explains sampling statements.

Stan Functions

real **lkj_corr_log**(matrix *y*, real *eta*)
The log of the LKJ density for the correlation matrix *y* given nonnegative shape *eta*.

matrix **lkj_corr_rng**(int *K*, real *eta*)
Generate a LKJ random correlation matrix of order *K* with shape *eta*; may only be used in generated quantities block

46. Covariance Matrix Distributions

The covariance matrix distributions have support on symmetric, positive-definite $K \times K$ matrices.

46.1. Wishart Distribution

Probability Density Function

If $K \in \mathbb{N}$, $\nu \in (K - 1, \infty)$, and $S \in \mathbb{R}^{K \times K}$ is symmetric and positive definite, then for symmetric and positive-definite $W \in \mathbb{R}^{K \times K}$,

$$\mathsf{Wishart}(W | \nu, S) = \frac{1}{2^{\nu K / 2}} \; \frac{1}{\Gamma_K\left(\frac{\nu}{2}\right)} \; |S|^{-\nu/2} \; |W|^{(\nu - K - 1)/2} \; \exp\left(\frac{1}{2} \, \mathrm{tr}\left(S^{-1} W\right)\right),$$

where $\mathrm{tr}()$ is the matrix trace function, and $\Gamma_K()$ is the multivariate Gamma function,

$$\Gamma_K(x) = \frac{1}{\pi^{K(K-1)/4}} \; \prod_{k=1}^{K} \Gamma\left(x + \frac{1 - k}{2}\right).$$

Sampling Statement

`W ~ `**`wishart`**`(nu, Sigma);`

Increment log probability with `wishart_log(W, nu, Sigma)`, dropping constant additive terms; Section 23.3 explains sampling statements.

Stan Functions

real **wishart_log**(matrix *W*, real *nu*, matrix *Sigma*)

The log of the Wishart density for positive-definite matrix *W* given degrees of freedom *nu* and scale matrix *Sigma*

matrix **wishart_rng**(real *nu*, matrix *Sigma*)

Generate a Wishart variate with degrees of freedom *nu* and scale matrix *Sigma*; may only be used in generated quantities block

46.2. Inverse Wishart Distribution

Probability Density Function

If $K \in \mathbb{N}$, $\nu \in (K-1, \infty)$, and $S \in \mathbb{R}^{K \times K}$ is symmetric and positive definite, then for symmetric and positive-definite $W \in \mathbb{R}^{K \times K}$,

$$\mathsf{InvWishart}(W|\nu, S) = \frac{1}{2^{\nu K/2}} \; \frac{1}{\Gamma_K\left(\frac{\nu}{2}\right)} \; |S|^{\nu/2} \; |W|^{-(\nu-K-1)/2} \; \exp\left(-\frac{1}{2} \; \mathrm{tr}(SW^{-1})\right).$$

Sampling Statement

```
W ~ inv_wishart(nu,Sigma);
```
Increment log probability with inv_wishart_log(*W*,*nu*,*Sigma*), dropping constant additive terms; Section 23.3 explains sampling statements.

Stan Functions

real **inv_wishart_log**(matrix *W*, real *nu*, matrix *Sigma*)

The log of the inverse Wishart density for positive-definite matrix *W* given degrees of freedom *nu* and scale matrix *Sigma*

matrix **inv_wishart_rng**(real *nu*, matrix *Sigma*)

Generate an inverse Wishart variate with degrees of freedom *nu* and scale matrix *Sigma*; may only be used in generated quantities block

46.3. LKJ Covariance Distribution

Sampling Statement

```
W ~ lkj_cov_log(mu,sigma,eta);
```
Increment log probability with lkj_cov_log_log(*W*,*mu*,*sigma*,*eta*), dropping constant additive terms; Section 23.3 explains sampling statements.

Stan Functions

real **lkj_cov_log**(matrix *W*, vector *mu*, vector *sigma*, real *eta*)

The log of the LKJ density for covariance matrix *W* is the sum of log of the lognormal density of the standard deviations given location vector *mu* and scale vector *sigma* and the log of the lkj_corr density of the correlation matrix given shape *eta*. See the next section for details on the lkj_corr density.

Part VIII

Additional Topics

47.   Bayesian Data Analysis

Gelman et al. (2003) provide the following characterization of Bayesian data analysis.

> By Bayesian data analysis, we mean practical methods for making inferences from data using probability models for quantities we observe and about which we wish to learn.

They go on to describe how Bayesian statistics differs from frequentist approaches.

> The essential characteristic of Bayesian methods is their explicit use of probability for quantifying uncertainty in inferences based on statistical analysis.

Because they view probability as the limit of relative frequencies of observations, strict frequentists forbid probability statements about parameters. Parameters are considered fixed, not random.

Bayesians also treat parameters as fixed but unknown. But unlike frequentists, they make use of both prior distributions over parameters and posterior distributions over parameters. These prior and posterior probabilities and posterior predictive probabilities are intended to characterize knowledge about the parameters and future observables. Posterior distributions form the basis of Bayesian inference, as described below.

47.1.   Bayesian Modeling

(Gelman et al., 2003) break applied Bayesian modeling into the following three steps.

1. Set up a full probability model for all observable and unobservable quantities. This model should be consistent with existing knowledge of the data being modeled and how it was collected.

2. Calculate the posterior probability of unknown quantities conditioned on observed quantities. The unknowns may include unobservable quantities such as parameters and potentially observable quantities such as predictions for future observations.

3. Evaluate the model fit to the data. This includes evaluating the implications of the posterior.

Typically, this cycle will be repeated until a sufficient fit is achieved in the third step. Stan automates the calculations involved in the second and third steps.

47.2. Bayesian Inference

Basic Quantities

The mechanics of Bayesian inference follow directly from Bayes's rule. To fix notation, let y represent observed quantities such as data and let θ represent unknown quantities such as parameters and future observations. Both y and θ will be modeled as random. Let x represent known, but unmodeled quantities such as constants, hyperparameters, and predictors.

Probability Functions

The probability function $p(y, \theta)$ is the joint probability function of the data y and parameters θ. The constants and predictors x are implicitly understood as being part of the conditioning. The conditional probability function $p(y|\theta)$ of the data y given parameters θ and constants x is called the sampling probability function; it is also called the likelihood function when viewed as a function of θ for fixed y and x.

The probability function $p(\theta)$ over the parameters given the constants x is called the prior because it characterizes the probability of the parameters before any data is observed. The conditional probability function $p(\theta|y)$ is called the posterior because it characterizes the probability of parameters given observed data y and constants x.

Bayes's Rule

The technical apparatus of Bayesian inference hinges on the following chain of equations, known in various forms as Bayes's rule (where again, the constants x are implicit).

$$
\begin{aligned}
p(\theta|y) \quad &= \quad \frac{p(\theta, y)}{p(y)} && \text{[definition of conditional probability]} \\[2ex]
&= \quad \frac{p(y|\theta)\,p(\theta)}{p(y)} && \text{[chain rule]} \\[2ex]
&= \quad \frac{p(y|\theta)\,p(\theta)}{\int_\Theta p(y, \theta)\,d\theta} && \text{[law of total probability]} \\[2ex]
&= \quad \frac{p(y|\theta)\,p(\theta)}{\int_\Theta p(y|\theta)\,p(\theta)\,d\theta} && \text{[chain rule]} \\[2ex]
&\propto \quad p(y|\theta)\,p(\theta) && \text{[y is fixed]}
\end{aligned}
$$

Bayes's rule "inverts" the probability of the posterior $p(\theta|y)$, expressing it solely in terms of the likelihood $p(y|\theta)$ and prior $p(\theta)$ (again, with constants and predictors

x implicit). The last step is important for Stan, which only requires probability functions to be characterized up to a constant multiplier.

Predictive Inference

The uncertainty in the estimation of parameters θ from the data y (given the model) is characterized by the posterior $p(\theta|y)$. The posterior is thus crucial for Bayesian predictive inference.

If $\tilde{y}$ is taken to represent new, perhaps as yet unknown, observations, along with corresponding constants and predictors $\tilde{x}$, then the posterior predictive probability function is given by

$$p(\tilde{y}|y) = \int_{\Theta} p(\tilde{y}|\theta)\,p(\theta|y)\,d\theta.$$

Here, both the original constants and predictors x and the new constants and predictors $\tilde{x}$ are implicit. Like the posterior itself, predictive inference is characterized probabilistically. Rather than using a point estimate of the parameters θ, predictions are made based on averaging the predictions over a range of θ weighted by the posterior probability $p(\theta|y)$ of θ given data y (and constants x).

The posterior may also be used to estimate event probabilities. For instance, the probability that a parameter θ_k is greater than zero is characterized probabilistically by

$$\Pr[\theta_k > 0] = \int_{\Theta} \mathrm{I}(\theta_k > 0)\,p(\theta|y)\,d\theta.$$

The indicator function, $\mathrm{I}(\phi)$, evaluates to one if the proposition ϕ is true and evaluates to zero otherwise.

Comparisons involving future observables may be carried out in the same way. For example, the probability that $\tilde{y}_n > \tilde{y}_{n'}$ can be characterized using the posterior predictive probability function as

$$\Pr[\tilde{y}_n > \tilde{y}_{n'}] = \int_{\Theta}\int_{Y} \mathrm{I}(\tilde{y}_n > \tilde{y}_{n'})\,p(\tilde{y}|\theta)p(\theta|y)\,d\tilde{y}\,d\theta.$$

Posterior Predictive Checking

After the parameters are fit to data, they can be used to simulate a new data set by running the model inferences in the forward direction. These replicated data sets can then be compared to the original data either visually or statistically to assess model fit (Gelman et al., 2003, Chapter 6).

In Stan, posterior simulations can be generated in two ways. The first approach is to treat the predicted variables as parameters and then define their distributions in the model block. The second approach, which also works for discrete variables, is to

generate replicated data using random-number generators in the generated quantities block.

48.   Markov Chain Monte Carlo Sampling

Like BUGS, Stan uses Markov chain Monte Carlo (MCMC) techniques to generate samples from the posterior distribution for inference.

48.1.   Monte Carlo Sampling

Monte Carlo methods were developed to numerically approximate integrals that are not tractable analytically but for which evaluation of the function being integrated is tractable (Metropolis and Ulam, 1949).

For example, the mean μ of a probability density $p(\theta)$ is defined by the integral

$$\mu = \int_{\Theta} \theta \times p(\theta) \, d\theta.$$

For even a moderately complex Bayesian model, the posterior density $p(\theta|y)$ leads to an integral that is impossible to evaluate analytically. The posterior also depends on the constants and predictors x, but from here, they will just be elided and taken as given.

Now suppose it is possible to draw independent samples from $p(\theta)$ and let $\theta^{(1)}, \theta^{(2)}, \ldots, \theta^{(N)}$ be N such samples. A Monte Carlo estimate $\hat{\mu}$ of the mean μ of $p(\theta)$ is given by the sample average,

$$\hat{\mu} = \frac{1}{N} \sum_{n=1}^{N} \theta^{(n)}.$$

If the probability function $p(\theta)$ has a finite mean and variance, the law of large numbers ensures the Monte Carlo estimate converges to the correct value as the number of samples increases,

$$\lim_{N \to \infty} \hat{\mu} = \mu.$$

Assuming finite mean and variance, estimation error is governed by the central limit theorem, so that estimation error decreases as the square root of N,

$$|\mu - \hat{\mu}| \propto \frac{1}{\sqrt{N}}.$$

Therefore, estimating a mean to an extra decimal place of accuracy requires one hundred times more samples; adding two decimal places means ten thousand times as many samples. This makes Monte Carlo methods more useful for rough estimates to within a few decimal places than highly precise estimates. In practical applications, there is no point estimating a quantity beyond the uncertainty of the data sample on which it is based, so this lack of many decimal places of accuracy is rarely a problem in practice for statistical models.

48.2.   Markov Chain Monte Carlo Sampling

Markov chain Monte Carlo (MCMC) methods were developed for situations in which it is not straightforward to draw independent samples (Metropolis et al., 1953).

A Markov chain is a sequence of random variables $\theta^{(1)}, \theta^{(2)}, \ldots$ where each variable is conditionally independent of all other variables given the value of the previous value. Thus if $\theta = \theta^{(1)}, \theta^{(2)}, \ldots, \theta^{(N)}$, then

$$p(\theta) = p(\theta^{(1)}) \prod_{n=2}^{N} p(\theta^{(n)} | \theta^{(n-1)}).$$

Stan generates a next state in a manner described in Section 48.5.

The Markov chains Stan and other MCMC samplers generate are ergodic in the sense required by the Markov chain central limit theorem, meaning roughly that there is there is a reasonable chance of reaching one value of θ from another. The Markov chains are also stationary, meaning that the transition probabilities do not change at different positions in the chain, so that for $n, n' \geq 0$, the probability function $p(\theta^{(n+1)} | \theta^{(n)})$ is the same as $p(\theta^{(n'+1)} | \theta^{(n')})$ (following the convention of overloading random and bound variables and picking out a probability function by its arguments).

Stationary Markov chains have an equilibrium distribution on states in which each has the same marginal probability function, so that $p(\theta^{(n)})$ is the same probability function as $p(\theta^{(n+1)})$. In Stan, this equilibrium distribution $p(\theta^{(n)})$ is the probability function $p(\theta)$ being sampled, typically a Bayesian posterior density.

Using MCMC methods introduces two difficulties that are not faced by independent sample Monte Carlo methods. The first problem is determining when a randomly initialized Markov chain has converged to its equilibrium distribution. The second problem is that the draws from a Markov chain are correlated, and thus the central limit theorem's bound on estimation error no longer applies. These problems are addressed in the next two sections.

48.3.   Initialization and Convergence Monitoring

A Markov chain generates samples from the target distribution only after it has converged to equilibrium. Unfortunately, this is only guaranteed in the limit in theory. In practice, diagnostics must be applied to monitor whether the Markov chain(s) have converged.

Potential Scale Reduction

One way to monitor whether a chain has converged to the equilibrium distribution is to compare its behavior to other randomly initialized chains. This is the motivation

for the [Gelman and Rubin](#) (1992) potential scale reduction statistic, $\hat{R}$. The $\hat{R}$ statistic measures the ratio of the average variance of samples within each chain to the variance of the pooled samples across chains; if all chains are at equilibrium, these will be the same and $\hat{R}$ will be one. If the chains have not converged to a common distribution, the $\hat{R}$ statistic will be greater than one.

Gelman and Rubin's recommendation is that the independent Markov chains be initialized with diffuse starting values for the parameters and sampled until all values for $\hat{R}$ are below 1.1. Stan allows users to specify initial values for parameters and it is also able to draw diffuse random initializations itself.

The $\hat{R}$ statistic is defined for a set of M Markov chains, θ_m, each of which has N samples $\theta_m^{(n)}$. The between-sample variance estimate is

$$B = \frac{N}{M-1} \sum_{m=1}^{M} (\bar{\theta}_m^{(\bullet)} - \bar{\theta}_\bullet^{(\bullet)})^2,$$

where

$$\bar{\theta}_m^{(\bullet)} = \frac{1}{N} \sum_{n=1}^{N} \theta_m^{(n)} \quad \text{and} \quad \bar{\theta}_\bullet^{(\bullet)} = \frac{1}{M} \sum_{m=1}^{M} \bar{\theta}_m^{(\bullet)}.$$

The within-sample variance is

$$W = \frac{1}{M} \sum_{m=1}^{M} s_m^2,$$

where

$$s_m^2 = \frac{1}{N-1} \sum_{n=1}^{N} (\theta_m^{(n)} - \bar{\theta}_m^{(\bullet)})^2.$$

The variance estimator is

$$\widehat{\text{var}}^+(\theta|y) = \frac{N-1}{N} W + \frac{1}{N} B.$$

Finally, the potential scale reduction statistic is defined by

$$\hat{R} = \sqrt{\frac{\widehat{\text{var}}^+(\theta|y)}{W}}.$$

Generalized $\hat{R}$ for Ragged Chains

Now suppose that each chain may have a different number of samples. Let N_m be the number of samples in chain m. Now the formula for the within-chain mean for chain m uses the size of the chain, N_m,

$$\bar{\theta}_m^{(\bullet)} = \frac{1}{N_m} \sum_{n=1}^{N} \theta_n^{(m)},$$

as does the within-chain variance estimate,

$$s_m^2 = \frac{1}{N_m - 1} \sum_{n=1}^{N_m} (\theta_m^{(n)} - \bar{\theta}_m^{(\bullet)})^2.$$

The terms that average over chains, such as $\bar{\theta}_{\bullet}^{(\bullet)}$, B, and W, have the same definition as before to ensure that each chain has the same effect on the estimate. If the averages were weighted by size, a single long chain would dominate the statistics and defeat the purpose of monitoring convergence with multiple chains.

Because it contains the term N, the estimate $\widehat{var}^+$ must be generalized. By expanding the first term,

$$\frac{N-1}{N} W = \frac{N-1}{N} \frac{1}{M} \sum_{m=1}^{M} \frac{1}{N-1} \sum_{n=1}^{N} (\theta_m^{(n)} - \bar{\theta}_m^{(\bullet)})^2 = \frac{1}{M} \sum_{m=1}^{M} \frac{1}{N} \sum_{n=1}^{N} (\theta_m^{(n)} - \bar{\theta}_m^{(\bullet)})^2,$$

and the second term,

$$\frac{1}{N} B = \frac{1}{M-1} \sum_{m=1}^{M} (\bar{\theta}_m^{(\bullet)} - \bar{\theta}_{\bullet}^{(\bullet)})^2.$$

the variance estimator naturally generalizes to

$$\widehat{var}^+(\theta|y) = \frac{1}{M} \sum_{m=1}^{M} \frac{1}{N_m} \sum_{n=1}^{N_m} (\theta_m^{(n)} - \bar{\theta}_m^{(\bullet)})^2 + \frac{1}{M-1} \sum_{m=1}^{M} (\bar{\theta}_m^{(\bullet)} - \bar{\theta}_{\bullet}^{(\bullet)})^2.$$

If the chains are all the same length, this definition is equivalent to the one in the last section. This generalized variance estimator and the within-chains variance estimates may be plugged directly into the formula for $\hat{R}$ from the previous section.

Split $\hat{R}$ for Detecting Non-Stationarity

Before calculating the potential-scale-reduction statistic $\hat{R}$, each chain may be split into two halves. This provides an additional means to detect non-stationarity in the chains. If one chain involves gradually increasing values and one involves gradually decreasing values, they have not mixed well, but they can have $\hat{R}$ values near unity. In this case, splitting each chain into two parts leads to $\hat{R}$ values substantially greater than 1 because the first half of each chain has not mixed with the second half.

48.4. Effective Sample Size

The second technical difficulty posed by MCMC methods is that the samples will typically be autocorrelated within a chain. This increases the uncertainty of the estimation of posterior quantities of interest, such as means, variances or quantiles.

Definition of Effective Sample Size

The amount by which autocorrelation within the chains increases uncertainty in estimates can be measured by effective sample size (ESS). Given independent samples, the central limit theorem bounds uncertainty in estimates based on the number of samples N. Given dependent samples, the number of independent samples is replaced with the effective sample size N_{eff}, which is the number of independent samples with the same estimation power as the N autocorrelated samples. For example, estimation error is proportional to $1/\sqrt{N_{\text{eff}}}$ rather than $1/\sqrt{N}$.

The effective sample size of a sequence is defined in terms of the autocorrelations within the sequence at different lags. The autocorrelation ρ_t at lag $t \geq 0$ for a chain with joint probability function $p(\theta)$ with mean μ and variance σ^2 is defined to be

$$\rho_t = \frac{1}{\sigma^2} \int_\Theta (\theta^{(n)} - \mu)(\theta^{(n+t)} - \mu) \, p(\theta) \, d\theta.$$

This is just the correlation between the two chains offset by t positions. Because we know $\theta^{(n)}$ and $\theta^{(n+t)}$ have the same marginal distribution in an MCMC setting, multiplying the two difference terms and reducing yields

$$\rho_t = \frac{1}{\sigma^2} \int_\Theta \theta^{(n)} \, \theta^{(n+t)} \, p(\theta) \, d\theta.$$

The effective sample size of N samples generated by a process with autocorrelations ρ_t is defined by

$$N_{\text{eff}} = \frac{N}{\sum_{t=-\infty}^{\infty} \rho_t} = \frac{N}{1 + 2 \sum_{t=1}^{\infty} \rho_t}.$$

Estimation of Effective Sample Size

In practice, the probability function in question cannot be tractably integrated and thus the autocorrelation cannot be calculated, nor the effective sample size. Instead, these quantities must be estimated from the samples themselves. The rest of this section describes a variogram-based estimator for autocorrelations, and hence effective sample size, based on multiple chains. For simplicity, each chain θ_m will be assumed to be of length N.

One way to estimate the effective sample size is based on the variograms V_t at lag $t \in \{0, 1 \ldots\}$. The variograms are defined as follows for (univariate) samples $\theta_m^{(n)}$, where $m \in \{1, \ldots, M\}$ is the chain, and N_m is the number of samples in chain m.

$$V_t = \frac{1}{M} \sum_{m=1}^{M} \left(\frac{1}{N_m} \sum_{n=t+1}^{N_m} \left(\theta_m^{(n)} - \theta_m^{(n-t)} \right)^2 \right).$$

The variogram along with the multi-chain variance estimate $\widehat{\text{var}}^+$ introduced in the previous section can be used to estimate the autocorrelation at lag t as

$$\hat{\rho}_t = 1 - \frac{V_t}{2\,\widehat{\text{var}}^+}.$$

If the chains have not converged, the variance estimator $\widehat{\text{var}}^+$ will overestimate variance, leading to an overestimate of autocorrelation and an underestimate effective sample size.

Because of the noise in the correlation estimates $\hat{\rho}_t$ as t increases, typically only the initial estimates of $\hat{\rho}_t$ where $\hat{\rho}_t < 0$ will be used. Setting T' to be the first lag such that $\rho_{T'+1} < 0$,

$$T' = \arg\min_t \ \hat{\rho}_{t+1} < 0,$$

the effective sample size estimator is defined as

$$\hat{N}_{\text{eff}} = \frac{MN}{1 + \sum_{t=1}^{T'} \hat{\rho}_t}.$$

Thinning Samples

In the typical situation, the autocorrelation, ρ_t, decreases as the lag, t, increases. When this happens, thinning the samples will reduce the autocorrelation. For instance, consider generating one thousand samples in one of the following two ways.

1. Generate 1000 samples after convergence and save all of them.

2. Generate 10,000 samples after convergence and save every tenth sample.

Even though both produce one thousand samples, the second approach with thinning will produce more effective samples. That's because the autocorrelation ρ_t for the thinned sequence is equivalent to ρ_{10t} in the unthinned samples, so the sum of the autocorrelations will be lower and thus the effective sample size higher.

On the other hand, if memory and data storage are no object, saving all ten thousand samples will have a higher effective sample size than thinning to one thousand samples.

48.5. Stan's Hamiltonian Monte Carlo Samplers

For continuous variables, Stan uses Hamiltonian Monte Carlo (HMC) sampling. HMC is a Markov chain Monte Carlo (MCMC) method based on simulating the Hamiltonian dynamics of a fictional physical system in which the parameter vector θ represents the position of a particle in K-dimensional space and potential energy is defined to

be the negative (unnormalized) log probability. Each sample in the Markov chain is generated by starting at the last sample, applying a random momentum to determine initial kinetic energy, then simulating the path of the particle in the field. Standard HMC runs the simulation for a fixed number of discrete steps of a fixed step size, whereas NUTS adjusts the number of steps on each iteration and allows varying step sizes per parameter.

Step-Size Adaptation during Warmup

In addition to standard HMC, Stan implements an adaptive version of HMC, the No-U-Turn Sampler (NUTS). By default, NUTS automatically tunes a step sizes during warmup. A global step size is optimized for a target Metropolis-Hastings reject rate using dual averaging; see (Nesterov, 2009) for details of dual averaging and (Hoffman and Gelman, 2011, 2013) for details of its use in Stan. For information on run-time configuration of step-size adaptation, see Section 4.3. Then step sizes per parameter are estimated during warmup.

Number of Steps

During sampling, NUTS adapts the number of leapfrog steps (i.e., the simulation time), using a geometric criterion that stops a trajectory when it begins to head back in the direction of the initial state. Once a trajectory is stopped, NUTS uses slice sampling to select a state along the trajectory as the next proposal.

Although Stan only samples continuous variables, its language is expressive enough to allow most discrete variables to be marginalized out; see Chapter 11 for examples.

Detailed Balance

HMC uses a Metropolis rejection step to ensure detailed balance of the resulting Markovian system; for details, see (Neal, 2011). NUTS uses a form of slice sampling which guarantees detailed balance; for details, see (Hoffman and Gelman, 2011, 2013).[1] This adjustment treats the momentum term of the Hamiltonian as an auxiliary variable, and the only reason for rejecting a sample will be discretization error

---

[1]A transition density $\phi(\omega'|\omega)$ and density $\pi(\omega)$ over state space Ω satisfy detailed balance if and only if for all $\omega, \omega' \in \Omega$,

$$\pi(\omega)\,\phi(\omega'|\omega) = \pi(\omega')\,\phi(\omega|\omega').$$

Detailed balance ensures stationarity of the transition density ϕ with respect to the equilibrium density π, so that

$$\pi(\omega') = \int_\Omega \pi(\omega)\,\phi(\omega'|\omega)\,d\omega.$$

in computing the Hamiltonian. In practice, the possibility of rejecting a proposed update means that one or more duplicate samples may appear in the chain; the resulting loss in inferential power is accounted for with effective sample size calculations as described in Section 48.4.

49.   Transformations of Variables

To avoid having to deal with constraints while simulating the Hamiltonian dynamics
during sampling, every (multivariate) parameter in a Stan model is transformed to an
unconstrained variable behind the scenes by the model compiler. The transform is
based on the constraints, if any, in the parameter's definition. Scalars or the scalar
values in vectors, row vectors or matrices may be constrained with lower and/or up-
per bounds. Vectors may alternatively be constrained to be ordered, positive ordered,
or simplexes. Matrices may be constrained to be correlation matrices or covariance
matrices. This chapter provides a definition of the transforms used for each type of
variable.

Stan converts models to C++ classes which define probability functions with sup-
port on all of $\mathbb{R}^K$, where K is the number of unconstrained parameters needed to
define the constrained parameters defined in the program. The C++ classes also in-
clude code to transform the parameters from unconstrained to constrained and apply
the appropriate Jacobians.

49.1.   Changes of Variables

The support of a random variable X with density $p_X(x)$ is that subset of values for
which it has non-zero density,

$$\text{supp}(X) = \{x | p_X(x) > 0\}.$$

If f is a total function defined on the support of X, then $Y = f(X)$ is a new random
variable. This section shows how to compute the probability density function of Y
for well-behaved transforms f. The rest of the chapter details the transforms used by
Stan.

Univariate Changes of Variables

Suppose X is one dimensional and $f : \text{supp}(X) \to \mathbb{R}$ is a one-to-one, monotonic func-
tion with a differentiable inverse f^{-1}. Then the density of Y is given by

$$p_Y(y) = p_X(f^{-1}(y)) \left| \frac{d}{dy} f^{-1}(y) \right|.$$

The absolute derivative of the inverse transform measures how the scale of the trans-
formed variable changes with respect to the underlying variable.

Multivariate Changes of Variables

The multivariate generalization of an absolute derivative is a Jacobian, or more fully the absolute value of the determinant of the Jacobian matrix of the transform. The Jacobian matrix measures the change of each output variable relative to every input variable and the absolute determinant uses that to determine the differential change in volume at a given point in the parameter space.

Suppose X is a K-dimensional random variable with probability density function $p_X(x)$. A new random variable $Y = f(X)$ may be defined by transforming X with a suitably well-behaved function f. It suffices for what follows to note that if f is one-to-one and its inverse f^{-1} has a well-defined Jacobian, then the density of Y is

$$p_Y(y) = p_X(f^{-1}(y)) \mid \det J_{f^{-1}}(y) \mid,$$

where $\det$ is the matrix determinant operation and $J_{f^{-1}}(y)$ is the Jacobian matrix of f^{-1} evaluated at y. Taking $x = f^{-1}(y)$, the Jacobian matrix is defined by

$$J_{f^{-1}}(y) = \begin{bmatrix} \dfrac{\partial x_1}{\partial y_1} & \cdots & \dfrac{\partial x_1}{\partial y_K} \\ \vdots & \vdots & \vdots \\ \dfrac{\partial x_K}{\partial y_1} & \cdots & \dfrac{\partial x_K}{\partial y_K} \end{bmatrix}.$$

If the Jacobian matrix is triangular, the determinant reduces to the product of the diagonal entries,

$$\det J_{f^{-1}}(y) = \prod_{k=1}^{K} \frac{\partial x_k}{\partial y_k}.$$

Triangular matrices naturally arise in situations where the variables are ordered, for instance by dimension, and each variable's transformed value depends on the previous variable's transformed values. Diagonal matrices, a simple form of triangular matrix, arise if each transformed variable only depends on a single untransformed variable.

49.2. Lower Bounded Scalar

Stan uses a logarithmic transform for lower and upper bounds.

Lower Bound Transform

If a variable X is declared to have lower bound a, it is transformed to an unbounded variable Y, where

$$Y = \log(X - a).$$

Lower Bound Inverse Transform

The inverse of the lower-bound transform maps an unbounded variable Y to a variable X that is bounded below by a by

$$X = \exp(Y) + a.$$

Absolute Derivative of the Lower Bound Inverse Transform

The absolute derivative of the inverse transform is

$$\left| \frac{d}{dy}\left(\exp(y) + a\right) \right| = \exp(y).$$

Therefore, given the density p_X of X, the density of Y is

$$p_Y(y) = p_X(\exp(y) + a) \cdot \exp(y).$$

49.3. Upper Bounded Scalar

Stan uses a negated logarithmic transform for upper bounds.

Upper Bound Transform

If a variable X is declared to have an upper bound b, it is transformed to the unbounded variable Y by

$$Y = \log(b - X).$$

Upper Bound Inverse Transform

The inverse of the upper bound transform converts the unbounded variable Y to the variable X bounded above by b through

$$X = b - \exp(Y).$$

Absolute Derivative of the Upper Bound Inverse Transform

The absolute derivative of the inverse of the upper bound transform is

$$\left| \frac{d}{dy}\left(b - \exp(y)\right) \right| = \exp(y).$$

Therefore, the density of the unconstrained variable Y is defined in terms of the density of the variable X with an upper bound of b by

$$p_Y(y) = p_X(b - \exp(y)) \cdot \exp(y).$$

49.4. Lower and Upper Bounded Scalar

For lower and upper-bounded variables, Stan uses a scaled and translated log-odds transform.

Log Odds and the Logistic Sigmoid

The log-odds function is defined for $u \in (0, 1)$ by

$$\text{logit}(u) = \log \frac{u}{1 - u}.$$

The inverse of the log odds function is the logistic sigmoid, defined for $v \in (-\infty, \infty)$ by

$$\text{logit}^{-1}(v) = \frac{1}{1 + \exp(-v)}.$$

The derivative of the logistic sigmoid is

$$\frac{d}{dy}\text{logit}^{-1}(y) = \text{logit}^{-1}(y) \cdot \left(1 - \text{logit}^{-1}(y)\right).$$

Lower and Upper Bounds Transform

For variables constrained to be in the open interval (a, b), Stan uses a scaled and translated log-odds transform. If variable X is declared to have lower bound a and upper bound b, then it is transformed to a new variable Y, where

$$Y = \text{logit}\left(\frac{X - a}{b - a}\right).$$

Lower and Upper Bounds Inverse Transform

The inverse of this transform is

$$X = a + (b - a) \cdot \text{logit}^{-1}(Y).$$

Absolute Derivative of the Lower and Upper Bounds Inverse Transform

The absolute derivative of the inverse transform is given by

$$\left| \frac{d}{dy}\left(a + (b - a) \cdot \text{logit}^{-1}(y)\right) \right| = (b - a) \cdot \text{logit}^{-1}(y) \cdot \left(1 - \text{logit}^{-1}(y)\right).$$

Therefore, the density of the transformed variable Y is

$$p_Y(y) = p_X\left(a + (b - a) \cdot \text{logit}^{-1}(y)\right) \cdot (b - a) \cdot \text{logit}^{-1}(y) \cdot \left(1 - \text{logit}^{-1}(y)\right).$$

Despite the apparent complexity of this expression, most of the terms are repeated and thus only need to be evaluated once. Most importantly, $\mathrm{logit}^{-1}(y)$ only needs to be evaluated once, so there is only one call to $\exp(-y)$.

49.5. Ordered Vector

For some modeling tasks, a vector-valued random variable X is required with support on ordered sequences. One example is the set of cut points in ordered logistic regression (see Section 12.6).

In constraint terms, an ordered K-vector $x \in \mathbb{R}^K$ satisfies

$$x_k < x_{k+1}$$

for $k \in \{1, \ldots, K-1\}$.

Ordered Transform

Stan's transform follows the constraint directly. It maps an increasing vector $x \in \mathbb{R}^K$ to an unconstrained vector $y \in \mathbb{R}^K$ by setting

$$y_k = \begin{cases} x_1 & \text{if } k = 1, \text{ and} \\ \log(x_k - x_{k-1}) & \text{if } 1 < k \leq K. \end{cases}$$

Ordered Inverse Transform

The inverse transform for an unconstrained $y \in \mathbb{R}^K$ to an ordered sequence $x \in \mathbb{R}^K$ is defined by the recursion

$$x_k = \begin{cases} y_1 & \text{if } k = 1, \text{ and} \\ x_{k-1} + \exp(y_k) & \text{if } 1 < k \leq K. \end{cases}$$

x_k can also be expressed iteratively as

$$x_k = y_1 + \sum_{k'=2}^{k} \exp(y_{k'}).$$

Absolute Jacobian Determinant of the Ordered Inverse Transform

The Jacobian of the inverse transform f^{-1} is lower triangular, with diagonal elements for $1 \leq k \leq K$ of

$$J_{k,k} = \begin{cases} 1 & \text{if } k = 1, \text{ and} \\ \exp(y_k) & \text{if } 1 < k \leq K. \end{cases}$$

Because J is triangular, the absolute Jacobian determinant is

$$| \det J | = \left| \prod_{k=1}^{K} J_{k,k} \right| = \prod_{k=2}^{K} \exp(y_k).$$

Putting this all together, if p_X is the density of X, then the transformed variable Y has density p_Y given by

$$p_Y(y) = p_X(f^{-1}(y)) \prod_{k=2}^{K} \exp(y_k).$$

49.6. Unit Simplex

Variables constrained to the unit simplex show up in multivariate discrete models as both parameters (categorical and multinomial) and as variates generated by their priors (Dirichlet and multivariate logistic).

The unit K-simplex is the set of points $x \in \mathbb{R}^K$ such that for $1 \le k \le K$,

$$x_k > 0,$$

and

$$\sum_{k=1}^{K} x_k = 1.$$

An alternative definition is to take the convex closure of the vertices. For instance, in 2-dimensions, the simplex vertices are the extreme values $(0, 1)$, and $(1, 0)$ and the unit 2-simplex is the line connecting these two points; values such as $(0.3, 0.7)$ and $(0.99, 0.01)$ lie on the line. In 3-dimensions, the basis is $(0, 0, 1)$, $(0, 1, 0)$ and $(1, 0, 0)$ and the unit 3-simplex is the boundary and interior of the triangle with these vertices. Points in the 3-simplex include $(0.5, 0.5, 0)$, $(0.2, 0.7, 0.1)$ and all other triplets of non-negative values summing to 1.

As these examples illustrate, the simplex always picks out a subspace of $K - 1$ dimensions from $\mathbb{R}^K$. Therefore a point x in the K-simplex is fully determined by its first $K - 1$ elements $x_1, x_2, \ldots, x_{K-1}$, with

$$x_K = 1 - \sum_{k=1}^{K-1} x_k.$$

Unit Simplex Inverse Transform

Stan's unit simplex inverse transform may be understood using the following stick-breaking metaphor.[1]

---

[1] For an alternative derivation of the same transform using hyperspherical coordinates, see (Betancourt, 2010).

Take a stick of unit length (i.e., length 1). Break a piece off and label it as x_1, and set it aside. Next, break a piece off what's left, label it x_2, and set it aside. Continue doing this until you have broken off $K - 1$ pieces labeled $(x_1, \ldots, x_{K-1})$. Label what's left of the original stick x_K. The vector $x = (x_1, \ldots, x_K)$ is obviously a unit simplex because each piece has non-negative length and the sum of their lengths is 1.

This full inverse mapping requires the breaks to be represented as the fraction in $(0, 1)$ of the original stick that is broken off. These break ratios are themselves derived from unconstrained values in $(-\infty, \infty)$ using the inverse logit transform as described above for unidimensional variables with lower and upper bounds.

More formally, an intermediate vector $z \in \mathbb{R}^{K-1}$, whose coordinates z_k represent the proportion of the stick broken off in step k, is defined elementwise for $1 \le k < K$ by

$$z_k = \text{logit}^{-1}\left(y_k + \log\left(\frac{1}{K - k}\right)\right).$$

The logit term $\log\left(\frac{1}{K-k}\right)$ $(i.e., \text{logit}\left(\frac{1}{K-k+1}\right))$ in the above definition adjusts the transform so that a zero vector y is mapped to the simplex $x = (1/K, \ldots, 1/K)$. For instance, if $y_1 = 0$, then $z_1 = 1/K$; if $y_2 = 0$, then $z_2 = 1/(K - 1)$; and if $y_{K-1} = 0$, then $z_{K-1} = 1/2$.

The break proportions z are applied to determine the stick sizes and resulting value of x_k for $1 \le k < K$ by

$$x_k = \left(1 - \sum_{k'=1}^{k-1} x_{k'}\right) z_k.$$

The summation term represents the length of the original stick left at stage k. This is multiplied by the break proportion z_k to yield x_k. Only $K - 1$ unconstrained parameters are required, with the last dimension's value x_K set to the length of the remaining piece of the original stick,

$$x_K = 1 - \sum_{k=1}^{K-1} x_k.$$

Absolute Jacobian Determinant of the Unit-Simplex Inverse Transform

The Jacobian J of the inverse transform f^{-1} is lower-triangular, with diagonal entries

$$J_{k,k} = \frac{\partial x_k}{\partial y_k} = \frac{\partial x_k}{\partial z_k}\frac{\partial z_k}{\partial y_k},$$

where

$$\frac{\partial z_k}{\partial y_k} = \frac{\partial}{\partial y_k}\text{logit}^{-1}\left(y_k + \log\left(\frac{1}{K - k}\right)\right) = z_k(1 - z_k),$$

and

$$\frac{\partial x_k}{\partial z_k} = \left(1 - \sum_{k'=1}^{k-1} x_{k'}\right).$$

This definition is recursive, defining x_k in terms of $x_1, \ldots, x_{k-1}$.

Because the Jacobian J of f^{-1} is lower triangular and positive, its absolute determinant reduces to

$$|\det J| = \prod_{k=1}^{K-1} J_{k,k} = \prod_{k=1}^{K-1} z_k (1 - z_k) \left(1 - \sum_{k'=1}^{k-1} x_{k'}\right).$$

Thus the transformed variable $Y = f(X)$ has a density given by

$$p_Y(y) = p_X(f^{-1}(y)) \prod_{k=1}^{K-1} z_k (1 - z_k) \left(1 - \sum_{k'=1}^{k-1} x_{k'}\right).$$

Even though it is expressed in terms of intermediate values z_k, this expression still looks more complex than it is. The exponential function need only be evaluated once for each unconstrained parameter y_k; everything else is just basic arithmetic that can be computed incrementally along with the transform.

Unit Simplex Transform

The transform $Y = f(X)$ can be derived by reversing the stages of the inverse transform. Working backwards, given the break proportions z, y is defined elementwise by

$$y_k = \text{logit}(z_k) - \log\left(\frac{1}{K - k}\right).$$

The break proportions z_k are defined to be the ratio of x_k to the length of stick left after the first $k - 1$ pieces have been broken off,

$$z_k = \frac{x_k}{1 - \sum_{k'=1}^{k-1} x_{k'}}.$$

49.7. Unit Vector

Unit vectors show up in directional statistics.

The n-sphere is the set of points $x \in \mathbb{R}^n$ such that

$$\|x\|^2 = \sum_{i=1}^{n} x_i^2 = 1 .$$

Unit Vector Inverse Transform

To parametrize unit length vectors, we use hyperspherical coordinates. The unconstrained vector $y \in \mathbb{R}^{n-1}$ is a set of angles which relates to the unit vector as follows:

$$
\begin{aligned}
x_1 &= \cos(y_1) \\
x_2 &= \cos(y_2)\sin(y_1) \\
x_3 &= \cos(y_3)\sin(y_1)\sin(y_2) \\
&\;\;\vdots \\
x_i &= \cos(y_i)\prod_{j=1}^{i-1}\sin(y_j) \\
&\;\;\vdots \\
x_n &= \prod_{j=1}^{n-1}\sin(y_j).
\end{aligned}
$$

Note that, in practice, we use $y_i = \hat{y}_i + \frac{\pi}{2}$ and use $\hat{y}_i$ as the unconstrained parameters to avoid a singularity at $y_i = 0$.

Absolute Jacobian Determinant of the Unit Vector Inverse Transform

To derive the determinant of the Jacobian we first add a radius coordinate $r = 1$ which multiplies the vector x. The Jacobian, J, can be shown to be lower triangular, so its determinant is just the product of diagonal entries and the absolute value of the determinant is

$$
|\det J| = \left| r^{n-1}\sin^{n-2}(y_1)\sin^{n-3}(y_2)\ldots\sin(y_{n-2}) \right|.
$$

49.8. Correlation Matrices

A $K \times K$ correlation matrix x must be is a symmetric, so that

$$
x_{k,k'} = x_{k',k}
$$

for all $k, k' \in \{1, \ldots, K\}$, it must have a unit diagonal, so that

$$
x_{k,k} = 1
$$

for all $k \in \{1, \ldots, K\}$, and it must be positive definite, so that for every non-zero K-vector a,

$$
a^\top x a > 0.
$$

To deal with this rather complicated constraint, Stan implements the transform of Lewandowski et al. (2009). The number of free parameters required to specify a $K \times K$ correlation matrix is $\binom{K}{2}$.

Correlation Matrix Inverse Transform

It is easiest to specify the inverse, going from its $\binom{K}{2}$ parameter basis to a correlation matrix. The basis will actually be broken down into two steps. To start, suppose y is a vector containing $\binom{K}{2}$ unconstrained values. These are first transformed via the bijective function $\tanh : \mathbb{R} \to (0, 1)$

$$\tanh x = \frac{\exp(2x) - 1}{\exp(2x) + 1}.$$

Then, define a $K \times K$ matrix z, the upper triangular values of which are filled by row with the transformed values. For example, in the 4×4 case, there are $\binom{4}{2}$ values arranged as

$$z = \begin{bmatrix} 0 & \tanh y_1 & \tanh y_2 & \tanh y_4 \\ 0 & 0 & \tanh y_3 & \tanh y_5 \\ 0 & 0 & 0 & \tanh y_6 \\ 0 & 0 & 0 & 0 \end{bmatrix}.$$

Lewandowski et al. show how to bijectively map the array z to a correlation matrix x. The entry $z_{i,j}$ for $i < j$ is interpreted as the canonical partial correlation (CPC) between i and j, which is the correlation between i's residuals and j's residuals when both i and j are regressed on all variables i' such that $i' < i$. In the case of $i = 1$, there are no earlier variables, so $z_{1,j}$ is just the Pearson correlation between i and j.

In Stan, the LKJ transform is reformulated in terms of a Cholesky factor w of the final correlation matrix, defined for $1 \leq i, j \leq K$ by

$$w_{i,j} = \begin{cases} 0 & \text{if } i > j, \\ 1 & \text{if } 1 = i = j, \\ \prod_{i'=1}^{i-1} \left(1 - z_{i',j}^2\right)^{1/2} & \text{if } 1 < i = j, \\ z_{i,j} & \text{if } 1 = i < j, \text{ and} \\ z_{i,j} \prod_{i'=1}^{i-1} \left(1 - z_{i',j}^2\right)^{1/2} & \text{if } 1 < i < j. \end{cases}$$

This does not require as much computation per matrix entry as it may appear; calcu-

lating the rows in terms of earlier rows yields the more manageable expression

$$w_{i,j} = \begin{cases} 0 & \text{if } i > j, \\ 1 & \text{if } 1 = i = j, \\ z_{i,j} & \text{if } 1 = i < j, \text{ and} \\ z_{i,j}\, w_{i-1,j} \left(1 - z_{i-1,j}^2\right)^{1/2} & \text{if } 1 < i \le j. \end{cases}$$

Given the upper-triangular Cholesky factor w, the final correlation matrix is

$$x = w^\top w.$$

Lewandowski et al. show that the determinant of the correlation matrix can be defined in terms of the canonical partial correlations as

$$\det x = \prod_{i=1}^{K-1} \prod_{j=i+1}^{K} (1 - z_{i,j}^2) = \prod_{1 \le i < j \le K} (1 - z_{i,j}^2),$$

Absolute Jacobian Determinant of the Correlation Matrix Inverse Transform

The only description so far is in the Stan transform code.

Correlation Matrix Transform

The correlation transform is defined by reversing the steps of the inverse transform defined in the previous section.

Starting with a correlation matrix x, the first step is to find the unique upper triangular w such that $x = ww^\top$. Because x is positive definite, this can be done by applying the Cholesky decomposition,

$$w = \text{chol}(x).$$

The next step from the Cholesky factor w back to the array z of CPCs is simplified by the ordering of the elements in the definition of w, which when inverted yields

$$z_{i,j} = \begin{cases} 0 & \text{if } i \le j, \\ w_{i,j} & \text{if } 1 = i < j, \text{ and} \\ w_{i,j} \prod_{i'=1}^{i-1} \left(1 - z_{i',j}^2\right)^{-2} & \text{if } 1 < i < j. \end{cases}$$

The final stage of the transform reverses the hyperbolic tangent transform, which is defined by

$$\tanh^{-1} v = \frac{1}{2} \log \left(\frac{1 + v}{1 - v}\right).$$

The inverse hyperbolic tangent function, $\tanh^{-1}$, is also called the Fisher transformation.

49.9. Covariance Matrices

A $K \times K$ matrix is a covariance matrix if it is symmetric and positive definite (see the previous section for definitions). It requires $K + \binom{K}{2}$ free parameters to specify a $K \times K$ covariance matrix.

Covariance Matrix Transform

Stan's covariance transform is based on a Cholesky decomposition composed with a log transform of the positive-constrained diagonal elements.[2]

If x is a covariance matrix (i.e., a symmetric, positive definite matrix), then there is a unique lower-triangular matrix $z = \mathrm{chol}(x)$ with positive diagonal entries, called a Cholesky factor, such that

$$x = z\, z^\top.$$

The off-diagonal entries of the Cholesky factor z are unconstrained, but the diagonal entries $z_{k,k}$ must be positive for $1 \leq k \leq K$.

To complete the transform, the diagonal is log-transformed to produce a fully unconstrained lower-triangular matrix y defined by

$$y_{m,n} = \begin{cases} 0 & \text{if } m < n, \\ \log z_{m,m} & \text{if } m = n, \text{ and} \\ z_{m,n} & \text{if } m > n. \end{cases}$$

Covariance Matrix Inverse Transform

The inverse transform reverses the two steps of the transform. Given an unconstrained lower-triangular $K \times K$ matrix y, the first step is to recover the intermediate

---

[2] An alternative to the transform in this section, which can be coded directly in Stan, is to parameterize a covariance matrix as a scaled correlation matrix. An arbitrary $K \times K$ covariance matrix Σ can be expressed in terms of a K-vector σ and correlation matrix Ω as

$$\Sigma = \mathrm{diag}(\sigma) \times \Omega \times \mathrm{diag}(\sigma),$$

so that each entry is just a deviation-scaled correlation,

$$\Sigma_{m,n} = \sigma_m \times \sigma_n \times \Omega_{m,n}.$$

matrix z by reversing the log transform,

$$z_{m,n} = \begin{cases} 0 & \text{if } m < n, \\ \exp(y_{m,m}) & \text{if } m = n, \text{ and} \\ y_{m,n} & \text{if } m > n. \end{cases}$$

The covariance matrix x is recovered from its Cholesky factor z by taking

$$x = z\,z^{\top}.$$

Absolute Jacobian Determinant of the Covariance Matrix Inverse Transform

The Jacobian is the product of the Jacobians of the exponential transform from the unconstrained lower-triangular matrix y to matrix z with positive diagonals and the product transform from the Cholesky factor z to x.

The transform from unconstrained y to Cholesky factor z has a diagonal Jacobian matrix, the absolute determinant of which is thus

$$\prod_{k=1}^{K} \frac{\partial}{\partial y_{k,k}} \exp(y_{k,k}) \;=\; \prod_{k=1}^{K} \exp(y_{k,k}) \;=\; \prod_{k=1}^{K} z_{k,k}.$$

The Jacobian matrix of the second transform from the Cholesky factor z to the covariance matrix x is also triangular, with diagonal entries corresponding to pairs (m, n) with $m \geq n$, defined by

$$\frac{\partial}{\partial z_{m,n}} (z\,z^{\top})_{m,n} \;=\; \frac{\partial}{\partial z_{m,n}} \left(\sum_{k=1}^{K} z_{m,k}\, z_{n,k} \right) \;=\; \begin{cases} 2\,z_{n,n} & \text{if } m = n \text{ and} \\ z_{n,n} & \text{if } m > n. \end{cases}$$

The absolute Jacobian determinant of the second transform is thus

$$2^{K} \prod_{m=1}^{K} \prod_{n=1}^{m} z_{n,n}.$$

Finally, the full absolute Jacobian determinant of the inverse of the covariance matrix transform from the unconstrained lower-triangular y to a symmetric, positive definite matrix x is the product of the Jacobian determinants of the exponentiation and product transforms,

$$2^{K} \left(\prod_{k=1}^{K} z_{k,k} \right) \left(\prod_{m=1}^{K} \prod_{n=1}^{m} z_{n,n} \right) \;=\; 2^{K} \prod_{k=1}^{K} z_{k,k}^{K-k+2}.$$

Let f^{-1} be the inverse transform from a $K + \binom{K}{2}$-vector y to the $K \times K$ covariance matrix x. A density function $p_X(x)$ defined on $K \times K$ covariance matrices is transformed to the density $p_Y(y)$ over $K + \binom{K}{2}$ vectors y by

$$p_Y(y) = p_X(f^{-1}(y)) \, 2^K \prod_{k=1}^{K} z_{k,k}^{K-k+2}.$$

49.10. Cholesky Factors of Covariance Matrices

An $M \times N$ matrix is a Cholesky factor of a covariance matrix if it is lower triangular, the diagonal entries are positive, and $M \geq N$. It requires $N + \binom{N}{2} + (M-N)N$ parameters.

Cholesky Factor of Covariance Matrix Transform

Stan's Cholesky factor transform only requires the first step of the covariance matrix transform, namely log transforming the positive diagonal elements. Suppose x is an $M \times N$ Cholesky factor. The above-diagonal entries are zero, the diagonal entries are positive, and the below-diagonal entries are unconstrained. The transform required is thus

$$y_{m,n} = \begin{cases} 0 & \text{if } m < n, \\ \log x_{m,m} & \text{if } m = n, \text{ and} \\ x_{m,n} & \text{if } m > n. \end{cases}$$

Cholesky Factor of Covariance Matrix Inverse Transform

The inverse transform need only invert the logarithm with an exponentiation. If y is the unconstrained matrix representation, then the elements of the constrained matrix x is defined by

$$x_{m,n} = \begin{cases} 0 & \text{if } m < n, \\ \exp(y_{m,m}) & \text{if } m = n, \text{ and} \\ y_{m,n} & \text{if } m > n. \end{cases}$$

Absolute Jacobian Determinant of Cholesky Factor Inverse Transform

The transform has a diagonal Jacobian matrix, the absolute determinant of which is

$$\prod_{n=1}^{N} \frac{\partial}{\partial y_{n,n}} \exp(y_{n,n}) = \prod_{n=1}^{N} \exp(y_{n,n}) = \prod_{n=1}^{N} x_{n,n}.$$

Let $x = f^{-1}(y)$ be the inverse transform from a $N + \binom{N}{2} + (M - N)N$ vector to an $M \times N$ Cholesky factor for a covariance matrix x defined in the previous section. A density function $p_X(x)$ defined on $M \times N$ Cholesky factors of covariance matrices is transformed to the density $p_Y(y)$ over $N + \binom{N}{2} + (M - N)N$ vectors y by

$$p_Y(y) = p_X(f^{-1}(y)) \prod_{N=1}^{N} x_{n,n}.$$

Part IX

Contributed Modules

50. Contributed Modules

Stan is an open-source project and welcomes user contributions.

In order to reduce maintenance on the main trunk of Stan development and to allow developer-specified licenses, contributed Stan modules are not distributed as part of Stan itself.

50.1. Contributing a Stan Module

Developers who have a Stan module to contribute should contact the Stan developers either through one of the following.

- stan-users mailing list:
  https://groups.google.com/forum/?fromgroups#!forum/stan-users

- Stan e-mail:
  mailto:stan@mc-stan.org

50.2. GitHub-Hosted Modules

The stan-dev organization on GitHub hosts contributed projects on GitHub. This is also where the Stan developers will host works in progress. The full list of contributed projects on GitHub for stan-dev is available at the following location.

>   https://github.com/stan-dev

Each contributed module on stan-dev's GitHub space comes with its own documentation, indexed by the README.md file displayed on GitHub. Each contributed module has its own licensing the terms of which are controlled by its developers. The license for a contributed package and its dependencies can be found in a top-level directory licenses/.

Emacs Stan Mode

Emacs Stan mode allows syntax highlighting and automatic indentation of Stan models in the Emacs text editor.

Repository:	https://github.com/stan-dev/stan-mode
License:	GPLv3
Authors:	Jeffrey Arnold, Daniel Lee

Appendices

A. Licensing

Stan and its two dependent libraries, Boost and Eigen, are distributed under liberal freedom-respecting[1] licenses approved by the Open Source Initiative.[2]

In particular, the licenses for Stan and its dependent libraries have no "copyleft" provisions requiring applications of Stan to be open source if they are redistributed.

This chapter describes the licenses for the tools that are distributed with Stan. The next chapter explains some of the build tools that are not distributed with Stan, but are required to build and run Stan models.

A.1. Stan's License

Stan is distributed under the BSD 3-clause license (BSD New).

    http://www.opensource.org/licenses/BSD-3-Clause

A.2. Boost License

Boost is distributed under the Boost Software License version 1.0.

    http://www.opensource.org/licenses/BSL-1.0

A.3. Eigen License

Eigen is distributed under the Mozilla Public License, version 2.

    http:/http://opensource.org/licenses/mpl-2.0

A.4. Google Test License

Stan uses Google Test for unit testing; it is not required to compile or execute models. Google Test is distributed under the BSD 2-clause license.

    http://www.opensource.org/licenses/BSD-License

---

[1]The link http://www.gnu.org/philosophy/open-source-misses-the-point.html leads to a discussion about terms "open source" and "freedom respecting."

[2]See http://opensource.org.

B.   Installation and Compatibility

This appendix describes the hardware and software required to run Stan. The software includes Stan and its libraries, as well as a contemporary C++ compiler. Stan requires hardware powerful enough to build and execute the models. Ideally, that will be a 64-bit computer with at least 4GB of memory and multiple processor cores.

B.1.   Operating System

Stan is written in portable C++ without C++11 features, as are the libraries on which it depends. Therefore, Stan should run on any machine for which a suitable C++ compiler is available. In practice, Stan, like the Boost and Eigen libraries on which it depends, is very hard on the compiler and linker.

Stan has been tested on the following operating systems.

- Linux (Debian, Ubuntu, Red Hat),

- Mac OS X (Snow Leopard, Lion, Mountain Lion), and

- Windows (XP, 7, 8).

Stan should work on other versions of these operating systems if compatible C++ compilers can be found. The plan is to keep up with new versions of these operating systems and gradually phase out testing on older versions.

B.2.   Step-by-Step Mac Install Instructions

This section provides step-by-step install instructions for the Mac; Linux and Windows sections follow. It repeats the step-by-step install instructions on Stan's home page at http://mc-stan.org/.

Stan has been tested on Mac OS X versions Snow Leopard, Lion, and Mountain Lion.

Tips for Mac Users

Finding and Opening Mac Applications and Files

To open an application, use [Command-Space] (press both keys at once on the keyboard) to open Spotlight, enter the application's name in the text field, then click on the application in the pop-up menu or [Return] if the right file or application is highlighted.

Spotlight can be used in the same way to find files or folders, such as the default `Downloads` folder for web downloads.

Open a Terminal for Shell Commands

To run shell commands, open the built-in Terminal application (see the previous subsection for details on how to find and open applications).

Install Xcode C++ Development Environment

The easiest (but not the only) way to install a C++ development environment on a Mac is to use Apple's Xcode development environment.

From the Xcode home page,

> https://developer.apple.com/xcode/

click `View in Mac App Store`.

From the App Store, click `Install`, enter an Apple ID, and wait for Xcode to finish installing.

Open the Xcode application, click top-level menu `Preferences`, click top-row button `Downloads`, click button for `Components`, click on the `Install` button to the right of the `Command Line Tools` entry, then wait for it to finish installing.

Click the top-level menu item `Xcode`, then click item `Quit Xcode` to quit.

To test, open the Terminal application and enter

```
> make --version
> g++ --version
```

Verify that `make` is at version 3.81 or later and `g++` is at 4.2.1 or later.

Download and Unpack Stan Source

Download the most recent version of `stan-2.m.p.tar.gz` (`m` is the minor version and `p` the patch level) from the Stan downloads list,

> https://github.com/stan-dev/stan/releases

Open the folder containing the download in the Finder (typically, the user's top-level `Downloads` folder).

If the Mac OS has not automatically unpacked the `.tar.gz` file into file `stan-2.m.p.tar`, double-click the `.tar.gz` file to unpack.

Double click on the `.tar` file to unarchive directory `stan-2.m.p`.

Move the resulting directory to a location where it will not be deleted, henceforth called `<stan-home>`.

B.3.   Step-by-Step Linux Install Instructions

Stan has been tested on various Linux installations, including Ubuntu, Debian, and Red Hat.

Installing C++ Development Tools

On Linux, C++ compilers and `make` are often installed by default.

To see if the `g++` compiler and `make` build system are already installed, use the commands

```
> g++ --version
```

and

```
> make --version
```

If these are at least at `g++` version 4.2.1 or later and `make` version 3.81 or later, no additional installations are necessary. It may still be desirable to update the C++ compiler `g++`, because later versions are faster.

To install the latest version of these tools (or upgrade an older version), use the commands

```
> sudo apt-get install g++
```

and

```
> sudo apt-get install make
```

A password will likely be required by the superuser command `sudo`.

Downloading and Unpacking Stan Source

Download the most recent stable version of Stan, `stan-2.m.p.tar.gz`, where `m` is the minor version and `p` the patch level), from the Stan downloads page,

https://github.com/stan-dev/stan/releases

to the directory where Stan will reside.

In a command shell, change directories to where the tarball was downloaded, say `<download-dir>`, with

```
> cd <download-dir>
```

where `<download-dir>` is replaced with the actual path to the directory.

Then, unpack the distribution into the subdirectory

```
        <download-dir>/stan-2.m.p
```

with

```
    > tar –xzf stan-2.m.p.tar.gz
```

B.4.   Step-by-Step Windows Install Instructions

Stan has been tested on Windows XP, Windows 7, and Windows 8.

Stan also runs under Cygwin, which provides a unix-like shell on top of Windows. Instructions for Cygwin installation are provided below in their own subsection.

Windows Tips

Opening a Command Shell

To open a Windows command shell, first open the `Start Menu` (usually in the lower left of the screen), select option `All Programs`, then option `Accessories`, then program `Command Prompt`.

Alternatively, enter [`Windows+r`] (both keys together on the keyboard), and enter `cmd` into the text field that pops up in the Run window, then press [`Return`] on the keyboard to run.

32-bit Builds

Stan defaults to a 64-bit build. On a 32-bit operating system, set the `BIT` variable to 32. For example, to build the Bernoulli model in Section 2.4, replace the original command with:

```
    > make BIT=32 src/models/basic_estimators/bernoulli
```

Rtools C++ Development Environment

The simplest way to install a full C++ build environment that will work for Stan is to use the Rtools package designed for R developers on Windows (even if you don't plan to use R).

First, download the latest *frozen* (i.e., stable) version of Rtools from the Rtools home page, using

http://cran.r-project.org/bin/windows/Rtools/

Next, double click on the downloaded file to open the Rtools install wizard, then proceed through its options.

- *Language*: select language, click `Next`,

- *Welcome*: click `Next`,

- *Information*: click `Next`,

- *Setup Location*: accept default (`c:\Rtools`), click `Next`,

- *Select Components*: select default, `Package Authoring`, click `Next`,

- *Select Additional Tasks*: check `Edit Path` and `Save Version in Registry`, click `Next`,

- *System Path Report*: click `Next`,

- *Ready to Install*: click `Next`, wait for the install to complete, then

- *Finish*: click `Finish`.

Downloading and Unpacking Stan

The Stan source code distributions are named `stan-2.m.p.tar.gz`, where `m` is the minor version and `p` the patch level.

Download the latest Stan source from the Stan downloads page,

https://github.com/stan-dev/stan/releases

to any non-temporary folder. (If in doubt, select `My Documents` on Windows XP or `Documents` on Windows 7.)

Change to the download directory (aka folder) using one of the following commands, replacing `<username>` with a Windows user name.

- *Windows XP*: From the default starting directory, use the following commands (quotes and all):

      > cd "My Documents"

  The full path (including quotes) will work from anywhere,

      > cd "c:\Documents and Settings\<username>\My Documents"

- *Windows 7*: From the default starting directory, use

      > cd Documents

  or use the full path, including quotes, from anywhere,

      > cd "c:\Users\<username>\Documents"

To verify that the downloaded Stan `.tar.gz` file is there, list the directory contents using:

```
> dir
```

Finally, unpack the distribution using the `tar` command (which is installed as part of Rtools).

```
> tar --no-same-owner -xzf stan-2.m.p.tar.gz
```

The `-no-same-owner` flag is not strictly necessary, but it removes a bunch of irrelevant warnings.

64-bit Cygwin Install Instructions

Stan can be run under Cygwin, the Unix look-and-feel environment for Windows. Cygwin must have recent versions of `make` and `g++` (part of gcc) installed. Within a Cygwin shell, Stan will behave as under other Unixes.

Thanks to Kevin van Horn for mailing the following instructions into the Stan-users mailing list. They only cover 64-bit R and 64-bit Cygwin, but that is what you should be using for Stan anyway.

1. Kill all Cygwin bash shells and shut down R.

2. After installing R and Rtools, make sure that R and Rtools are in the PATH environment variable. My R installation directory was `c:\Program Files\R\R-3.0.1` and my Rtools installation directory was `c:\Rtools`, so I added the following to the end of my user PATH variable:

   · `C:\Program Files\R\R-3.0.1\bin\x64`
   · `C:\Rtools\bin`
   · `C:\Rtools\gcc-4.6.3\bin`

3. Now there could be a conflict between Cygwin and Rtools when running a bash shell under Cygwin, so I added the following lines to my `.bash_profile` file to remove any PATH directory referencing Rtools:

   ```
   > TMP=`echo $PATH | /usr/bin/tr ':' '\n'                   \
                     | /usr/bin/egrep -iv '^/cygdrive/c/Rtools/' \
                     | tr '\n' ':'`
   > PATH=${TMP%:}
   ```

   (Note that the backslash characters signal that the line continues after a return.) This was only necessary to allow me to continue using Cygwin.

4. Apparently there is something in `Rcpp` or `inline` or `rstan` that doesn't like UNC paths. My home directory was `\\server\users\kevinv` and apparently this caused my local R library directory to be `\server\users\kevinv\R` as verified by `.libPaths()` from the R command prompt.

   I fixed this by copying the entire directory tree rooted at `\server\users\kevinv\R` over to a local directory on my workstation, `C:\Users\KevinV\R`, then adding the user environment variable `R_LIBS_USER=C:\Users\KevinV\R`. I shut down R and restarted it.

5. At this point the instructions given for installing Rstan finally worked.

B.5. Required Software and Tools

The only two absolute requirements for running Stan are the Stan source code (and dependent libraries) and a C++ compiler.

Stan Source

In order to compile Stan models, the Stan source code is required. The latest version of Stan can be downloaded from the following link.

> http://mc-stan.org/

The Stan source code distribution includes Stan's source code, documentation, build tools, unit tests, demo models, documentation and source for the required libraries Boost and Eigen, and the source for an optional testing library, Google Test.

Boost C++ Library Source

Stan's parser and some of its mathematical functions and template metaprogramming facilities are implemented with the Boost C++ Library.

- Home: http://www.boost.org/users/license.html

- License: Boost Software License

- Tested Version: 1.54.0

The Boost source code is distributed with Stan.

Eigen Matrix and Linear Algebra Library Source

Stan's matrix algebra depends on the Eigen C++ matrix and linear algebra library.

- · Home: http://eigen.tuxfamily.org
- · License: Mozilla Public License, version 2.0
- · Tested Version: 3.2.0

The Eigen source code is distributed with Stan.

C++ Compiler

Compiling Stan models requires a C++ compiler. Stan has been primarily developed with `clang++` and `g++` and no promises are made for other compilers. The full set of compilers for which Stan has been tested is

- · `g++`
  Tested Versions: Mac 4.2.1, 4.6, Linux 4.4–4.7 (plus trunk 4.8, 4.9), Windows 4.6.3
  Home: http://gcc.gnu.org/
  License: GPL3+
- · `clang++`, Mac 2.9–3.1, Linux 2.9–3.1
  Home: http://clang.llvm.org/
  License: BSD
- · mingw-64, version 2.0 (Windows 7, cross-compiled from Debian Linux)
- · Intel C++, Linux version 12.1.3

C++-11 Support

Stan 2.0 does not support C++-11. The remaining incompatibility with the parser will be included soon after Stan 2.0 is released. This will include support for the latest versions of `g++` and `clang++`.

B.6. Optional Components for Developers

Stan is developed using the following set of tools. The various command examples in this manual have assumed they can be found on the command path. The makefile allows precise locations to be plugged in.

GNU Make Build Tool

Stan automates the build, test, documentation, and deployment tasks using scripts in the form of makefiles to run with GNU Make.

- · Home: http://www.gnu.org/software/make

- · License: GPLv3+

- · Tested Versions: 3.81 (Mac OS X), 3.79 (Windows 7)

Doxygen Documentation Generator

Stan's API documentation is generated using the Doxygen Tool.

- · Home: http://www.stack.nl/~dimitri/doxygen/index.html

- · License: GPL2

- · Tested Version(s): Mac OS X 1.8.2, Windows 1.8.2

Git Version Control System

Stan uses the Git version control system for its software, libraries, and documentations. Git is required to interact with the most recent versions of code in the version control repository.

- · Home: http://git-scm.com/

- · License: GPL2

- · Tested Version(s): Mac version 1.7.8.4, Windows version 1.7.9

Google Test C++ Testing Framework

Stan's unit testing is based on the Google's googletest C++ testing framework.

- · Home: http://code.google.com/p/googletest/

- · License: BSD

- · Tested Version(s): 1.6.0

The Google Test framework is distributed with Stan.

B.7. Tips for Mac OS X

Install Xcode

Apple's Xcode contains both the `clang++` and `g++` compilers and make, all of the tools needed to work with Stan as a user. The version of Xcode to install depends on the version of Mac OS X.

Official Apple Xcode Distribution

Xcode 4 may be downloaded for free for Mac OS X 10.7 ("Lion") or later directly from Apple:

> Xcode 4: https://developer.apple.com/xcode/

Once you've installed Xcode, you need to start it, then open menu option `Xcode`, select `Preferences`, then click on the `Downloads` icon and then click on the `Install` button next to the option labeled "Command Line Tools."

At this point, you should have the make system `make` and the two C++ compilers/linkers, `g++` and `clang++`, installed. This is all you need to run Stan. Xcode will also install the `git` version control system at this point.

Alternative, GCC-Only Installer

A stripped down installer for just the GCC package, including the C++ compilers `g++` and `clang++`, available for Mac OS X 10.6 ("Snow Leopard") or later,

> https://github.com/kennethreitz/osx-gcc-installer/

The fill list of tools in this distribution is available at:

> http://www.opensource.apple.com/release/
> developer-tools-41/

More Recent Compilers

Alternative compilers to those distributed by Apple as part of Xcode are available at the following locations.

Homebrew

One way to get pre-built binaries for Mac OS X is to use Homebrew, which is available from the following link.

> http://mxcl.github.com/homebrew/

MacPorts

MacPorts hosts recent versions of compilers for the Macintosh.

> https://distfiles.macports.org/MacPorts/

After finding the appropriate `.dmg` file, clicking on it, then double clicking on the resulting `.pkg` file, and clicking through some more menus, the following will need to be entered from a terminal window to install it.

```
> sudo port install gccVersion
```

In this command, *gccVersion* is the name of a compiler version, such as `g++=mp-4.6`, for version 4.6. Errors may arise during the install such as the following.

```
Error:  Target org.macports.activate returned:  Image error:
/opt/local/include/gmp.h already exists and does not belong to
a registered port.  Unable to activate port gmp.  Use 'port -f
activate gmp' to force the activation.
```

This issue can be resolved by running the following command.

```
> sudo port -f activate gmp
```

Git Installer

A standalone version of Git for Mac OS X is available from the following site.

> http://code.google.com/p/git-osx-installer/

Although (at the time of this writing) there were only versions listed up to OS X version "Snow Leopard," they work on "Lion."

LaTeX Typesetting Package

Stan uses the LaTeX typesetting package for generating manuals, talks, and other materials (Doxygen is used for API documentation; see below). The first step is to download the MacTeX `.mpkg` file from the following URL [warning: the download is approximately 2GB and the installation approximately 3.5GB].

> http://www.tug.org/mactex/2011/

Once it is downloaded, just click on the `.mpkg` file and then follow the installer instructions. The installer will add the command to the `PATH` environment variable so that the `pdflatex` used by Stan is available from the command line.

Lucida Console Font

A free TrueType version of Lucida Console for the Mac is available at the following URL.

> `http://www.fontpalace.com/font-details/Lucida+Console/`

Download the `.ttf` file, then click on it to install. It will then be available as a preference in the Mac terminal application.

Doxygen API Documentation

Stan's API documentation is generated using the Doxygen tool. This tool is available from

> `http://www.doxygen.org`

Select the `Download` link from the second of the right-hand side navigation bars, then select the binary distribution `.dmg` file for Mac OS X. Clicking on the `.dmg` file opens the finder with a view of the unpacked Doxygen executable. Just drag the Doxygen icon into the Applications folder (or wherever you want to keep it). Then add the path to the Doxygen executable,

> `/Applications/Doxygen.app/Contents/Resources/doxygen`

to the system PATH environment variable. You can do add to the PATH environment by adding this line to the end of the top-level `~/.profile` file.

> `export PATH=/Applications/Doxygen.app/Contents/Resources:$PATH`

The next shell started will then be able to find the `doxygen` command.

B.8. Tips for Windows

Install Rtools

The easiest way to get a complete C++ build environment on Windows is to install the most recent version of Rtools.

The latest version verified to work with Stan is Rtools 2.15. Rtools 2.15 includes the g++ 4.6.3 (pre-release) compiler and many other useful command line tools including many Unix commands, such as the following.

```
basename, cat, cmp, comm, cp, cut, date, diff, du, echo,
expr, gzip, ls, make, makeinfo, mkdir, mv, rm, rsync, sed,
sh, sort, tar, texindex, touch, uniq
```

Rtools can be downloaded from the following location.

> http://cran.r-project.org/bin/windows/Rtools/

Install it using the Windows installer. Allow it to edit the PATH environment variable so that commands are available from the command tool.

To verify the installation was successful, open a command window by selecting the following menu items.

> Start → Accessories → Command Prompt

To verify that g++ is installed, use the following command.

> g++ -v

This should report version information for g++. Next, verify that make is installed with the following command.

> make -v

This should print version information for make.

Install Git

There are a number of Git clients for Windows that will work. The official Git installer for Windows can be found at the following location.

> http://code.google.com/p/msysgit/downloads

Select the latest full installer and install it.

C.   Stan for Users of BUGS

From the outside, Stan and BUGS[1] are similar — they use statistically-themed modeling languages (which are similar but with some differences; see below), they can be called from R, running some specified number of chains to some specified length, producing posterior simulations that can be assessed using standard convergence diagnostics. This is not a coincidence: in designing Stan, we wanted to keep many of the useful features of Bugs.

To start, take a look at the files of translated BUGS models at `http://mc-stan.org/`. These are 40 or so models from the BUGS example volumes, all translated and tested (to provide the same answers as BUGS) in Stan. For any particular model you want to fit, you can look for similar structures in these examples.

C.1.   Some Differences in How BUGS and Stan Work

· BUGS is interpreted; Stan is compiled in two steps, first a model is is translated to templated C++ and then to a platform-specific executable. Stan, unlike BUGS, allows the user to directly program in C++, but we do not describe how to do this in this Stan manual (see the getting started with C++ section of `http://mc-stan.org` for more information on using Stan directly from C++).

· BUGS performs MCMC updating one scalar parameter at a time (with some exceptions such as JAGS's implementation of regression and generalized linear models and some conjugate multivariate parameters), using conditional distributions (Gibbs sampling) where possible and otherwise using adaptive rejection sampling, slice sampling, and Metropolis jumping. BUGS figures out the dependence structure of the joint distribution as specified in its modeling language and uses this information to compute only what it needs at each step. Stan moves in the entire space of all the parameters using Hamiltonian Monte Carlo (more precisely, the no-U-turn sampler), thus avoiding some difficulties that occur with one-dimension-at-a-time sampling in high dimensions but at the cost of requiring the computation of the entire log density at each step.

· BUGS tunes its adaptive jumping (if necessary) during its warmup phase (traditionally referred to as "burn-in"). Stan uses its warmup phase to tune the no-U-turn sampler (NUTS).

---

[1]Except where otherwise noted, we use "BUGS" to refer to WinBUGS, OpenBUGS, and JAGS, indiscriminately.

- The BUGS modeling language is not directly executable. Rather, BUGS parses its model to determine the posterior density and then decides on a sampling scheme. In contrast, the statements in a Stan model are directly executable: they translate exactly into C++ code that is used to compute the log posterior density (which in turn is used to compute the gradient).

- In BUGS, the order in which statements are written does not matter. They are executed according to the directed graphical model so that variables are always defined when needed. A side effect of the direct execution of Stan's modeling language is that statements execute in the order in which they are written. For instance, the following Stan program, which sets mu before using it to sample y.

```
mu <- a + b * x;
y ~ normal(mu,sigma);
```

It translates to the following C++ code.

```
mu = a + b * x;
lp += normal_log(mu,sigma);
```

Contrast this with the Stan program

```
y ~ normal(mu,sigma)
mu <- a + b * x
```

This program is well formed, but is almost certainly a coding error, because it attempts to use mu before it is set. It translates to the following C++ code.

```
lp += normal_log(mu,sigma);
mu = a + b * x;
```

The direct translation to the imperative language of C++ code highlights the potential error of using mu in the first statement.

To trap these kinds of errors, variables are initialized to the special not-a-number (NaN) value. If NaN is passed to a log probability function, it will raise a domain exception, which will in turn be reported by the sampler. The sampler will reject the sample out of hand as if it had zero probability.

- Stan uses its own C++ algorithmic differentiation packages to compute the gradient of the log density (up to a proportion). Gradients are required during the Hamiltonian dynamics simulations within the leapfrog algorithm of the Hamiltonian Monte Carlo and NUTS samplers. BUGS computes the log density but not its gradient.

- Both BUGS and Stan are semi-automatic in that they run by themselves with no outside tuning required. Nevertheless, the user needs to pick the number of chains and number of iterations per chain. We usually pick 4 chains and start with 10 iterations per chain (to make sure there are no major bugs and to approximately check the timing), then go to 100, 1000, or more iterations as necessary. Compared to Gibbs or Metropolis, Hamiltonian Monte Carlo can take longer per iteration (as it typically takes many "leapfrog steps" within each iteration), but the iterations typically have lower autocorrelation. So Stan might work fine with 1000 iterations in an example where BUGS would require 100,000 for good mixing. We recommend monitoring potential scale reduction statistics ($\hat{R}$) and the effective sample size to judge when to stop (stopping when $\hat{R}$ values do not counter-indicate convergence and when enough effective samples have been collected).

- WinBUGS is closed source. OpenBUGS and JAGS are both licensed under the Gnu Public License (GPL), otherwise known as copyleft due to the restrictions it places on derivative works. Stan is licensed under the much more liberal new BSD license.

- Like WinBUGS, OpenBUGS and JAGS, Stan can be run directly from the command line or through R (Python and MATLAB interfaces are in the works)

- Like OpenBUGS and JAGS, Stan can be run on Linux, Mac, and Windows platforms.

C.2. Some Differences in the Modeling Languages

- The BUGS modeling language follows an R-like syntax in which line breaks are meaningful. Stan follows the rules of C, in which line breaks are equivalent to spaces, and each statement ends in a semicolon. For example:

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

and

```
for (i in 1:n) y[i] ~ normal(mu, sigma);
```

Or, equivalently (recall that a line break is just another form of whitespace),

```
for (i in 1:n)
  y[i] ~ normal(mu, sigma);
```

and also equivalently,

```
for (i in 1:n) {
  y[i] ~ normal(mu, sigma);
}
```

There's a semicolon after the model statement but not after the brackets indicating the body of the for loop.

· Another C thing: In Stan, variables can have names constructed using letters, numbers, and the underscore (_) symbol, but nothing else (and a variable name cannot begin with a number). BUGS variables can also include the dot, or period (.) symbol.

· In Stan, the second argument to the "normal" function is the standard deviation (i.e., the scale), not the variance (as in *Bayesian Data Analysis*) and not the inverse-variance (i.e., precision) (as in BUGS). Thus a normal with mean 1 and standard deviation 2 is `normal(1,2)`, not `normal(1,4)` or `normal(1,0.25)`.

· Similarly, the second argument to the "multivariate normal" function is the covariance matrix and not the inverse covariance matrix (i.e., the precision matrix) (as in BUGS). The same is true for the "multivariate student" distribution.

· The distributions have slightly different names:

BUGS	Stan
dnorm	normal
dbinom	binomial
dpois	poisson
⋮	⋮

· Stan, unlike BUGS, allows intermediate quantities, in the form of local variables, to be reassigned. For example, the following is legal and meaningful (if possibly inefficient) Stan code.

```
{
  total <- 0;
  for (i in 1:n){
    theta[i] ~ normal(total, sigma);
    total <- total + theta[i];
  }
}
```

In BUGS, the above model would not be legal because the variable `total` is defined more than once. But in Stan, the loop is executed in order, so `total` is overwritten in each step.

- Stan uses explicit declarations. Variables are declared with base type integer or real, and vectors, matrices, and arrays have specified dimensions. When variables are bounded, we give that information also. For data and transformed parameters, the bounds are used for error checking. For parameters, the constraints are critical to sampling as they determine the geometry over which the Hamiltonian is simulated.

  Variables can be declared as data, transformed data, parameters, transformed parameters, or generated quantities. They can also be declared as local variables within blocks. For more information, see the part of this manual devoted to the Stan programming language and examine at the example models.

- Stan allows all sorts of tricks with vector and matrix operations which can make Stan models more compact. For example, arguments to probability functions may be vectorized,[2] allowing

  ```
  for (i in 1:n)
    y[i] ~ normal(mu[i], sigma[i]);
  ```

  to be expressed more compactly as

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

  The vectorized form is also more efficient because Stan can unfold the computation of the chain rule during algorithmic differentiation.

- Stan also allows for arrays of vectors and matrices. For example, in a hierarchical model might have a vector of K parameters for each of J groups; this can be declared using

  ```
  vector[K] theta[J];
  ```

  Then theta[j] is an expression denoting a K-vector and may be used in the code just like any other vector variable.

  An alternative encoding would be with a two-dimensional array, as in

  ```
  real theta[J,K];
  ```

  The vector version can have some advantages, both in convenience and in computational speed for some operations.

  A third encoding would use a matrix:

---

[2]Most distributions have been vectorized, but currently the truncated versions may not exist and may not be vectorized.

```
matrix[J,K] theta;
```

but in this case, `theta[j]` is a row vector, not a vector, and accessing it as a vector is less efficient than with an array of vectors. The transposition operator, as in `theta[j]'`, may be used to convert the row vector `theta[j]` to a (column) vector. Column vector and row vector types are not interchangeable everywhere in Stan; see the function signature declarations in the programming language section of this manual.

· Stan supports general conditional statements using a standard if-else syntax. For example, a zero-inflated (or -deflated) Poisson mixture model of the form defined by Lambert (1992) may be defined as follows, where there is a probability θ of drawing a zero, and a probability $1 - \theta$ of drawing from $\mathsf{Poisson}(\lambda)$. The probability function is thus

$$p(y_n|\theta,\lambda) = \left\{ \begin{array}{ll} \theta + (1 - \theta) \times \mathsf{Poisson}(0|\lambda) & \text{if } y_n = 0, \text{ and} \\ (1 - \theta) \times \mathsf{Poisson}(y_n|\lambda) & \text{if } y_n > 0. \end{array} \right.$$

The log probability function can be implemented directly in Stan as follows.

```
data {
  int<lower=0> N;
  int<lower=0> y[N];
  ...
}
model {
  for (n in 1:N) {
    if (y[n] == 0)
      increment_log_prob(log_sum_exp(bernoulli_log(1,theta),
                                     bernoulli_log(0,theta)
                                     + poisson_log(y[n],lambda)));
    else
      increment_log_prob(bernoulli_Log(0,theta)
                         + poisson_log(y[n],lambda));
  }
  ...
}
```

The `log_sum_exp(lp1,lp2)` function adds the log probabilities on the linear scale; it is defined to be equal to `log(exp(lp1) + exp(lp2))`, but is more arithmetically stable.

· Stan supports general while loops using a standard syntax. While loops give Stan full Turing equivalent computational power. They are useful for defining

iterative functions with complex termination conditions. As an illustration of their syntax, the for-loop

```
model {
    ....
    for (n in 1:N) {
        ... do something with n ....
    }
}
```

may be recoded using the following while loop.

```
model {
    int n;
    ...
    n <- 1;
    while (n <= N) {
        ... do something with n ...
        n <- n + 1;
    }
}
```

C.3.   Some Differences in the Statistical Models that are Allowed

· Stan does not yet support sampling discrete parameters (discrete data is supported). We plan to implement discrete sampling using a combination of Gibbs and slice sampling but we haven't done so yet.

· Stan has some distributions on covariance matrices that do not exist in BUGS, including a uniform distribution over correlation matrices which may be rescaled, and the priors based on C-vines defined in (Lewandowski et al., 2009). In particular, the Lewandowski et al. prior allows the correlation matrix to be shrunk toward the unit matrix while the scales are given independent priors.

· In BUGS you need to define all variables. In Stan, if you declare but don't define a parameter it implicitly has a flat prior (on the scale in which the parameter is defined). For example, if you have a parameter p declared as

```
real<lower=0,upper=1> p;
```

and then have no sampling statement for p in the model block, then you are implicitly assigning a uniform $[0, 1]$ prior on p. On the other hand, if you have a parameter theta declared with

```
real theta;
```

and have no sampling statement for `theta` in the `model` block, then you are implicitly assigning an improper uniform prior on $(-\infty, \infty)$ to `theta`.

· BUGS models are always proper (being constructed as a product of proper marginal and conditional densities). Stan models can be improper. Here is the simplest improper Stan model:

```
parameters {
  real theta;
}
model { }
```

· Although parameters in Stan models may have improper priors, we do not want improper *posterior* distributions, as we are trying to use these distributions for Bayesian inference. There is no general way to check if a posterior distribution is improper. But if all the priors are proper, the posterior will be proper also.

· As noted earlier, each statement in a Stan model is directly translated into the C++ code for computing the log posterior. Thus, for example, the following pair of statements is legal in a Stan model:

```
y ~ normal(0,1);
y ~ normal(2,3);
```

The second line here does *not* simply overwrite the first; rather, *both* statements contribute to the density function that is evaluated. The above two lines have the effect of including the product, $\mathsf{Norm}(y|0, 1) \times \mathsf{Norm}(y|2, 3)$, into the density function.

For a perhaps more confusing example, consider the following two lines in a Stan model:

```
x ~ normal(0.8*y, sigma);
y ~ normal(0.8*x, sigma);
```

At first, this might look like a joint normal distribution with a correlation of 0.8. But it is not. The above are *not* interpreted as conditional entities; rather, they are factors in the joint density. Multiplying them gives, $\mathsf{Norm}(x|0.8y, \sigma) \times \mathsf{Norm}(y|0.8x, \sigma)$, which is what it is (you can work out the algebra) but it is not the joint distribution where the conditionals have regressions with slope 0.8.

· With censoring and truncation, Stan uses the censored-data or truncated-data likelihood—this is not always done in BUGS. All of the approaches to censoring and truncation discussed in (Gelman et al., 2003) and (Gelman and Hill, 2007) may be implemented in Stan directly as written.

- Stan, like BUGS, can benefit from human intervention in the form of reparameterization. More on this topic to come.

C.4. Some Differences when Running from R

- Stan can be set up from within R using two lines of code. Follow the instructions for running Stan from R on `http://mc-stan.org/`. You don't need to separately download Stan and RStan. Installing RStan will automatically set up Stan. When RStan moves to CRAN, it will get even easier.

- In practice we typically run the same Stan model repeatedly. If you pass RStan the result of a previously fitted model the model will not need be recompiled. An example is given on the running Stan from R pages available from `http://mc-stan.org/`.

- When you run Stan, it saves various conditions including starting values, some control variables for the tuning and running of the no-U-turn sampler, and the initial random seed. You can specify these values in the Stan call and thus achieve exact replication if desired. (This can be useful for debugging.)

- When running BUGS from R, you need to send exactly the data that the model needs. When running RStan, you can include extra data, which can be helpful when playing around with models. For example, if you remove a variable x from the model, you can keep it in the data sent from R, thus allowing you to quickly alter the Stan model without having to also change the calling information in your R script.

- As in R2WinBUGS and R2jags, after running the Stan model, you can quickly summarize using `plot()` and `print()`. You can access the simulations themselves using various extractor functions, as described in the RStan documentation.

- Various information about the sampler, such as number of leapfrog steps, log probability, and step size, is available through extractor functions. These can be useful for understanding what is going wrong when the algorithm is slow to converge.

C.5. The Stan Community

- Stan, like WinBUGS, OpenBUGS, and JAGS, has an active community, which you can access via the user's mailing list and the developer's mailing list; see

`http://mc-stan.org/` for information on subscribing and posting and to look at archives.

D.   Stan Program Style Guide

This appendix describes the preferred style for laying out Stan models. These are not rules of the language, but simply recommendations for laying out programs in a text editor. Although these recommendations may seem arbitrary, they are similar to those of many teams for many programming languages. Like rules for typesetting text, the goal is to achieve readability without wasting white space either vertically or horizontally.

D.1.   Choose a Consistent Style

The most important point of style is consistency. Consistent coding style makes it easier to read not only a single program, but multiple programs. So when departing from this style guide, the number one recommendation is to do so consistently.

D.2.   Line Length

Line lengths should not exceed 80 characters.[1] This is a typical recommendation for many programming language style guides because it makes it easier to lay out text edit windows side by side and to view the code on the web without wrapping, easier to view diffs from version control, etc. About the only thing that is sacrificed is laying out expressions on a single line.

D.3.   File Extensions

The recommended file extension for Stan model files is `.stan`. For Stan data dump files, the recommended extension is `.R`, or more informatively, `.data.R`.

D.4.   Variable Naming

The recommended variable naming is to follow C/C++ naming conventions, in which variables are lowercase, with the underscore character (_) used as a separator. Thus it is preferred to use `sigma_y`, rather than the run together `sigmay`, camel-case `sigmaY`, or capitalized camel-case `SigmaY`. Even matrix variables should be lowercased.

---

[1]Even 80 characters may be too many for rendering in print; for instance, in this manual, the number of code characters that fit on a line is about 65.

The exception to the lowercasing recommendation, which also follows the C/C++ conventions, is for size constants, for which the recommended form is a single uppercase letter. The reason for this is that it allows the loop variables to match. So loops over the indices of an $M \times N$ matrix a would look as follows.

```
for (m in 1:M)
  for (n in 1:N)
    a[m,n] = ...
```

D.5. Local Variable Scope

Declaring local variables in the block in which they are used aids in understanding programs because it cuts down on the amount of text scanning or memory required to reunite the declaration and definition.

The following Stan program corresponds to a direct translation of a BUGS model, which uses a different element of mu in each iteration.

```
model {
  real mu[N];
  for (n in 1:N) {
    mu[n] <- alpha * x[n] + beta;
    y[n] ~ normal(mu[n],sigma);
  }
}
```

Because variables can be reused in Stan and because they should be declared locally for clarity, this model should be recoded as follows.

```
model {
  for (n in 1:N) {
    real mu;
    mu <- alpha * x[n] + beta;
    y[n] ~ normal(mu,sigma);
  }
}
```

The local variable can be eliminated altogether, as follows.

```
model {
  for (n in 1:N)
    y[n] ~ normal(alpha * x[n] + beta, sigma);
}
```

There is unlikely to be any measurable efficiency difference between the last two implementations, but both should be a bit more efficient than the BUGS translation.

Scope of Compound Structures with Componentwise Assignment

In the case of local variables for compound structures, such as arrays, vectors, or matrices, if they are built up component by component rather than in large chunks, it can be more efficient to declare a local variable for the structure outside of the block in which it is used. This allows it to be allocated once and then reused.

```
model {
  vector[K] mu;
  for (n in 1:N) {
    for (k in 1:K)
      mu[k] <- ...;
    y[n] ~ multi_normal(mu,Sigma);
  }
}
```

In this case, the vector `mu` will be allocated outside of both loops, and used a total of N times.

D.6. Parentheses and Brackets

Optional Parentheses for Single-Statement Blocks

Single-statement blocks can be rendered in one of two ways. The fully explicit bracketed way is as follows.

```
for (n in 1:N) {
  y[n] ~ normal(mu,1);
}
```

The following statement without brackets has the same effect.

```
for (n in 1:N)
  y[n] ~ normal(mu,1);
```

Single-statement blocks can also be written on a single line, as in the following example.

```
for (n in 1:N) y[n] ~ normal(mu,1);
```

These can be much harder to read than the first example. Only use this style if the statement is very simple, as in this example. Unless there are many similar cases, it's almost always clearer to put each sampling statement on its own line.

Conditional and looping statements may also be written without brackets.

The use of for loops without brackets can be dangerous. For instance, consider this program.

```
for (n in 1:N)
  z[n] ~ normal(nu,1);
  y[n] ~ normal(mu,1);
```

Because Stan ignores whitespace and the parser completes a statement as eagerly as possible (just as in C++), the previous program is equivalent to the following program.

```
for (n in 1:N) {
  z[n] ~ normal(nu,1);
}
y[n] ~ normal(mu,1);
```

Parentheses in Nested Operator Expressions

The preferred style for operators minimizes parentheses. This reduces clutter in code that can actually make it harder to read expressions. For example, the expression `a + b * c` is preferred to the equivalent `a + (b * c)` or `(a + (b * c))`. The operator precedences and associativities are given in Figure 22.1.

Similarly, comparison operators can usually be written with minimal bracketing, with the form `y[n] > 0 || x[n] != 0` preferred to the bracketed form `(y[n] > 0) || (x[n] != 0)`.

No Open Brackets on Own Line

Vertical space is valuable as it controls how much of a program you can see. The preferred Stan style is as shown in the previous section, not as follows.

```
for (n in 1:N)
{
  y[n] ~ normal(mu,1);
}
```

This also goes for parameters blocks, transformed data blocks, which should look as follows.

```
transformed parameters {
  real sigma;
  ...
}
```

D.7. Conditionals

Stan supports the full C++-style conditional syntax, allowing real or integer values to act as conditions, as follows.

```
real x;
...
if (x) {
   // executes if x not equal to 0
   ...
}
```

Explicit Comparisons of Non-Boolean Conditions

The preferred form is to write the condition out explicitly for integer or real values that are not produced as the result of a comparison or boolean operation, as follows.

```
if (x != 0) ...
```

D.8.   White Space

Stan allows spaces between elements of a program. The white space characters allowed in Stan programs include the space (ASCII 0x20), line feed (ASCII 0x0A), carriage return (0x0D), and tab (0x09). Stan treats all whitespace characters interchangeably, with any sequence of whitespace characters being syntactically equivalent to a single space character. Nevertheless, effective use of whitespace is the key to good program layout.

Line Breaks Between Statements and Declarations

It is dispreferred to have multiple statements or declarations on the same line, as in the following example.

```
transformed parameters {
  real mu_centered;  real sigma;
  mu <- (mu_raw - mean_mu_raw);    sigma <- pow(tau,-2);
}
```

These should be broken into four separate lines.

No Tabs

Stan programs should not contain tab characters. They are legal and may be used anywhere other whitespace occurs. Using tabs to layout a program is highly unportable because the number of spaces represented by a single tab character varies depending on which program is doing the rendering and how it is configured.

Two-Character Indents

Stan has standardized on two space characters of indentation, which is the standard convention for C/C++ code. Another sensible choice is four spaces, which is the convention for Java and Python. Just be consistent.

Space Between `if` and Condition

Use a space after `if`s. For instance, use `if (x < y) ...`, not `if(x < y) ...`.

No Space For Function Calls

There is no space between a function name and the function it applies to. For instance, use `normal(0,1)`, not `normal (0,1)`.

Spaces Around Operators

There should be spaces around binary operators. For instance, use `y[1] <- x`, not `y[1]<-x`, use `(x + y) * z` not `(x+y)*z`.

Breaking Expressions across Lines

Sometimes expressions are too long to fit on a single line. In that case, the recommended form is to break *before* an operator,[2] aligning the operator to indicate scoping. For example, use the following form (though not the content; inverting matrices is almost always a bad idea).

```
increment_log_prob((y - mu)' * inv(Sigma) * (y - mu));
```

Here, the multiplication operator (`*`) is aligned to clearly signal the multiplicands in the product.

For function arguments, break after a comma and line the next argument up underneath as follows.

```
y[n] ~ normal(alpha + beta * x + gamma * y,
              pow(tau,-0.5));
```

___

[2]This is the usual convention in both typesetting and other programming languages. Neither R nor BUGS allows breaks before an operator because they allow newlines to signal the end of an expression or statement.

Optional Spaces after Commas

Optionally use spaces after commas in function arguments for clarity. For example, `normal(alpha * x[n] + beta,sigma)` can also be written as `normal(alpha * x[n] + beta, sigma)`.

Unix Newlines

Wherever possible, Stan programs should use a single line feed character to separate lines. All of the Stan developers (so far, at least) work on Unix-like operating systems and using a standard newline makes the programs easier for us to read and share.

Platform Specificity of Newlines

Newlines are signaled in Unix-like operating systems such as Linux and Mac OS X with a single line-feed (LF) character (ASCII code point `0x0A`). Newlines are signaled in Windows using two characters, a carriage return (CR) character (ASCII code point `0x0D`) followed by a line-feed (LF) character.

Bibliography

Betancourt, M. (2010). Cruising the simplex: Hamiltonian Monte Carlo and the Dirichlet distribution. *arXiv*, 1010(3436):1–5. 342

Betancourt, M. (2012). A general metric for Riemannian manifold Hamiltonian Monte Carlo. *arXiv*, 1212(4693). 40, 157

Betancourt, M. and Stein, L. C. (2011). The geometry of Hamiltonian Monte Carlo. *arXiv*, 1112(4118):1–9. 40

Blei, D. M. and Lafferty, J. D. (2007). A correlated topic model of *Science*. *The Annals of Applied Statistics*, 1(1):17–37. 131

Blei, D. M., Ng, A. Y., and Jordan, M. I. (2003). Latent Dirichlet allocation. *Journal of Machine Learning Research*, 3:993–1022. 129

Bowling, S. R., Khasawneh, M. T., Kaewkuekool, S., and Cho, B. R. (2009). A logistic approximation to the cumulative normal distribution. *Journal of Industrial Engineering and Management*, 2(1):114–127. 256

Clayton, D. G. (1992). Models for the analysis of cohort and case-control studies with inaccurately measured exposures. In Dwyer, J. H., Feinleib, M., Lippert, P., and Hoffmeister, H., editors, *Statistical Models for Longitudinal Studies of Exposure and Health*, pages 301–331. Oxford University Press. 117, 118

Cook, S. R., Gelman, A., and Rubin, D. B. (2006). Validation of software for Bayesian models using posterior quantiles. *Journal of Computational and Graphical Statistics*, 15(3):675–692. 69

Curtis, S. M. (2010). BUGS code for item response theory. *Journal of Statistical Software*, 31. 96

Daumé, III, H. (2007). HBC: Hierarchical Bayes compiler. Technical report, University of Utah. vi

Duane, A., Kennedy, A., Pendleton, B., and Roweth, D. (1987). Hybrid Monte Carlo. *Physics Letters B*, 195(2):216–222. 3

Durbin, J. and Koopman, S. J. (2001). *Time Series Analysis by State Space Methods*. Oxford University Press, New York. 319

Efron, B. (2012). *Large-Scale Inference: Empirical Bayes Methods for Estimation, Testing, and Prediction*. Institute of Mathematical Statistics Monographs. Cambridge Univesity Press. 6

Engle, R. F. (1982). Autoregressive conditional heteroscedasticity with estimates of variance of United Kingdom inflation. *Econometrica*, 50:987–1008. 102

Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D. (1999). *Refactoring: Improving the Design of Existing Code.* Addison-Wesley. ix

Gay, D. (2005). Semiautomatic differentiation for efficient gradient computations. In Bücker, H. M., Corliss, G. F., Hovland, P., Naumann, U., and Norris, B., editors, *Automatic Differentiation: Applications, Theory, and Implementations*, volume 50 of *Lecture Notes in Computational Science and Engineering*. Springer, New York. vii

Gelman, A., Carlin, J. B., Stern, H. S., and Rubin, D. B. (2003). *Bayesian Data Analysis.* CRC Press, London, 2nd edition. 6, 119, 121, 129, 147, 206, 325, 327, 376

Gelman, A. and Hill, J. (2007). *Data Analysis Using Regression and Multilevel-Hierarchical Models.* Cambridge University Press, Cambridge, United Kingdom. vi, 96, 97, 98, 129, 228, 376

Gelman, A. and Rubin, D. B. (1992). Inference from iterative simulation using multiple sequences. *Statistical Science*, 7(4):457–472. 5, 331

Giesler, G. C. (2000). MCNP software quality: Then and now. Technical Report LA-UR-00-2532, Los Alamos National Laboratory. xii

Girolami, M. and 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. 40, 157

Google (2011). Google Test: Google C++ testing framework. http://code.google.com/p/googletest/.

Guennebaud, G. and Jacob, B. (2012). Eigen C++ library, version 3.2. http://eigen.tuxfamily.org/.

Hastings, W. K. (1970). Monte Carlo sampling methods using Markov chains and their applications. *Biometrika*, 57(1):97–109. 4

Hoffman, M. D. and Gelman, A. (2011). The no-U-turn sampler: Adaptively setting path lengths in Hamiltonian Monte Carlo. *arXiv*, 1111(4246). viii, 4, 37, 40, 109, 178, 335

Hoffman, M. D. and Gelman, A. (2013). The no-U-turn sampler: Adaptively setting path lengths in Hamiltonian Monte Carlo. *Journal of Machine Learning Research*, in press. viii, 4, 37, 40, 109, 178, 335

Hunt, A. and Thomas, D. (1999). *The Pragmatic Programmer*. Addison-Wesley. 67

Kim, S., Shephard, N., and Chib, S. (1998). Stochastic volatility: Likelihood inference and comparison with ARCH models. *Review of Economic Studies*, 65:361–393. 108

Lambert, D. (1992). Zero-inflated Poisson regression, with an application to defects in manufacturing. *Technometrics*, 34(1). 374

Lewandowski, D., Kurowicka, D., and Joe, H. (2009). Generating random correlation matrices based on vines and extended onion method. *Journal of Multivariate Analysis*, 100:1989–2001. 321, 346, 375

McConnell, S. (2004). *Code Complete: A Practical Handbook of Software Construction*. Microsoft Press, second edition. 67

Metropolis, N., Rosenbluth, A., Rosenbluth, M., Teller, M., and Teller, E. (1953). Equations of state calculations by fast computing machines. *Journal of Chemical Physics*, 21:1087–1092. 4, 330

Metropolis, N. and Ulam, S. (1949). The Monte Carlo method. *Journal of the American Statistical Association*, 44(247):335–341. xii, 329

Neal, R. (2011). MCMC using Hamiltonian dynamics. In Brooks, S., Gelman, A., Jones, G. L., and Meng, X.-L., editors, *Handbook of Markov Chain Monte Carlo*, pages 116–162. Chapman and Hall/CRC. 3, 4, 41, 52, 335

Neal, R. M. (1994). An improved acceptance procedure for the hybrid monte carlo algorithm. *Journal of Computational Physics*, 111:194–203. 3

Neal, R. M. (1996). *Bayesian Learning for Neural Networks*. Number 118 in Lecture Notes in Statistics. Springer. 139

Neal, R. M. (1997). Monte Carlo implementation of Gaussian process models for Bayesian regression and classification. Technical Report 9702, University of Toronto, Department of Statistics. 139

Neal, R. M. (2003). Slice sampling. *Annals of Statistics*, 31(3):705–767. 155

Nesterov, Y. (2009). Primal-dual subgradient methods for convex problems. *Mathematical Programming*, 120(1):221–259. 4, 55, 335

Nocedal, J. and Wright, S. J. (2006). *Numerical Optimization*. Springer-Verlag, Berlin, second edition. 55

Papaspiliopoulos, O., Roberts, G. O., and SkoÌLld, M. (2007). A general framework for the parametrization of hierarchical models. *Statistical Science*, 22(1):59–73. 155

Pinheiro, J. C. and Bates, D. M. (1996). Unconstrained parameterizations for variance-covariance matrices. *Statistics and Computing*, 6:289–296.

Plummer, M., Best, N., Cowles, K., and Vines, K. (2006). CODA: Convergence diagnosis and output analysis for MCMC. *R News*, 6(1):7–11.

R Development Core Team (2012). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0.

Rasmussen, C. E. and Williams, C. K. I. (2006). *Gaussian Processes for Machine Learning*. MIT Press. 134

Richardson, S. and Gilks, W. R. (1993). A Bayesian approach to measurement error problems in epidemiology using conditional independence models. *American Journal of Epidemiology*, 138(6):430–442. 117

Rubin, D. B. (1981). Estimation in parallel randomized experiments. *Journal of Educational Statistics*, 6:377–401. 121

Schäling, B. (2011). The Boost C++ libraries. http://www.boost.org/.

Smith, T. C., Spiegelhalter, D. J., and Thomas, A. (1995). Bayesian approaches to random-effects meta-analysis: a comparative study. *Statistics in Medicine*, 14(24):2685–2699. 121

Tokuda, T., Goodrich, B., Mechelen, I. V., Gelman, A., and Tuerlinckx, F. (2010). Visualizing distributions of covariance matrices. Technical report, Columbia University, Department of Statistics.

van Heesch, D. (2011). Doxygen: Generate documentation from source code. http://www.stack.nl/~dimitri/doxygen/index.html.

Warn, D. E., Thompson, S. G., and Spiegelhalter, D. J. (2002). Bayesian random effects meta-analysis of trials with binary outcomes: methods for the absolute risk difference and relative risk scales. *Statistics in Medicine*, 21:1601–1623. 119, 121

Zyczkowski, K. and Sommers, H. (2001). Induced measures in the space of mixed quantum states. *Journal of Physics A: Mathematical and General*, 34(35):7111. 166

Index