

Stan Modeling Language

User's Guide and Reference Manual

Stan Development Team

Stan Version 1.1.0

2013-01-02



<http://mc-stan.org/>

Stan Development Team. 2012. *Stan Modeling Language: User's Guide and Reference Manual*. Version 1.1.0

Copyright © 2011–2012, Stan Development Team.

This document is distributed under the Creative Commons Attribute 3.0 Unported License (CC BY 3.0). For full details, see

[http:
//creativecommons.org/licenses/by/3.0/legalcode](http://creativecommons.org/licenses/by/3.0/legalcode)

Contents

Preface	v
Acknowledgements	ix
I Introduction	1
1. Overview	2
2. Getting Started	5
II Commands and Data Formats	11
3. Compiling Stan Programs	12
4. Running a Stan Program	20
5. Dump Data Format	24
III Programming Techniques	29
6. Print Statements and Debugging	30
7. Missing Data & Partially Known Parameters	33
8. Truncated or Censored Data	36
9. Mixture Modeling	41
10. Regression Models	44
11. Reparameterizations and Changes of Variables	56
12. Custom Probability Functions	61
13. Optimizing Stan Code	64
IV Modeling Language Reference	78
14. Execution of a Stan Program	79
15. Data Types and Variable Declarations	83
16. Expressions	94
17. Statements	104
18. Program Blocks	116
19. Modeling Language Syntax	126

V	Built-In Functions	129
20.	Integer-Valued Basic Functions	130
21.	Real-Valued Basic Functions	132
22.	Array Operations	141
23.	Matrix Operations	142
24.	Discrete Probabilities	153
25.	Continuous Probabilities	157
VI	Additional Topics	165
26.	Bayesian Data Analysis	166
27.	Markov Chain Monte Carlo Sampling	169
28.	Transformations of Variables	176
	Appendices	189
A.	Licensing	189
B.	Installation and Compatibility	190
C.	Stan for Users of BUGS	201
D.	Stan Program Style Guide	210
E.	Auxiliary Stan Tools	217

Preface

Why Stan?

We¹ 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 (?). 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 (?), 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.

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

¹In 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 University of Toronto, 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.

explored existing algorithmic differentiation packages with open licenses such as RAD (?) 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 1.0, 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 transforms.

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) (?), 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 is beyond the NUTS paper on *arXiv*, so we've been calling it NUTS II. 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.

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 kept at the top level of the release in the file `TO-DO.txt`.

You Can Help

Please let us know if you have comments on this document 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.

<http://code.google.com/p/stan/issues/list>

One of the main reasons Stan is freedom-respecting, open-source software² is that we love to collaborate. We're interested in hearing from you if you'd like to volunteer to get

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

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
2013-01-02

Acknowledgements

Institutions

We thank Columbia University along with the Departments of Statistics and Political Science, the Applied Statistics Center, the Institute for Social and Economic Research and Policy (ISERP), and the Core Research Computing Facility.

Grants

Stan was supported in part by the U. S. Department of Energy (DE-SC0002099), the U. S. National Science Foundation ATM-0934516 “Reconstructing Climate from Tree Ring Data.” and the U. S. Department of Education Institute of Education Sciences (ED-GRANTS-032309-005: “Practical Tools for Multilevel Hierarchical Modeling in Education Research” and R305D090006-09A: “Practical solutions for missing data”). The high-performance computing facility on which we ran evaluations was made possible through a grant from the U. S. National Institutes of Health (1G20RR030893-01: “Research Facility Improvement Grant”).

Stan is currently supported in part by a grant from the National Science Foundation (CNS-1205516)

Individuals

We thank John Salvatier for pointing us to automatic differentiation and HMC in the first place, and Kristen van Leuven of ISERP for help preparing our grant proposals.

We thank the following people for feedback on and corrections to the manual: Jeffrey Arnold, Eric N. Brown, Devin Caughey, Zhenming Su.

Code and Doc Patches

Thanks to Jeffrey Arnold for submitting code patches and reporting a whole bunch of bugs for earlier versions.

Thanks to Jeffrey Arnold, Eric N. Brown, Wayne Foltz, and Mike Ross for submitting documentation patches.

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://code.google.com/p/stan/issues/list>

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

Image from (?).



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 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.

1.2. 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 ex-

¹A 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.

ecutable for a model first reads in and validates the known values y and x , then generates a sequence of (non-independent) identically distributed samples $\theta^{(1)}, \theta^{(2)}, \dots$, each of which has the marginal distribution $p(\theta|y; x)$.

1.3. Stan’s Samplers

For continuous parameters, Stan uses Hamiltonian Monte Carlo (HMC) sampling (??), a form of Markov chain Monte Carlo (MCMC) sampling (?). Stan 1.0 can only sample discrete parameters on which no other parameters depend, such as simulated values. Chapter 9 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 (??).

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 tuning parameter is a mass matrix for the fictional particle. Stan can be configured to use a unit mass matrix or to estimate a diagonal mass matrix during warmup; in the future, it will also support a user-defined diagonal mass matrix. Estimating the mass matrix normalizes the scale of each element θ_k of the unknown variable sequence θ .

The other two tuning parameters set the temporal step size of the discretization of the Hamiltonian and the total number of steps taken per iteration. Stan can be configured with a user-specified step size or it can estimate an optimal step size during warmup using dual averaging (??). In either case, additional randomization may be applied to draw the step size from an interval of possible step sizes (?).

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 (?).

1.4. 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} (?).

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.

1.5. 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.

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
```

and then make the model parser and code generator with the following call on Unix-like systems (i.e., linux, Mac):

```
% make bin/stanc
```

The following variant is required for Windows:

```
% make bin/stanc.exe
```

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

Building `libstan.a` and `bin/stanc` need only be done 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 $y[1], \dots, y[N]$ are i.i.d. with Bernoulli chance-of-success θ . The prior on θ is $\text{beta}(1, 1)$ (i.e., uniform).¹

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 5 for more information).

¹If no prior were specified in the model block, the constraints on θ ensure it falls between 0 and 1, providing θ an implicit uniform prior. For parameters with no prior specified and unbounded support, the result is an improper prior. Stan accepts improper priors, but posteriors must be proper in order for sampling to succeed.

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 for Unix-like (Linux or Mac) operating systems:

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

and the following command in Windows:

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

And yes, those are forward slashes in the `make` target for Windows.

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.

Executing the Model

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

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

To execute the model under Linux or Mac, use

```
> ./bernoulli --data=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 --data=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.

```

STAN SAMPLING COMMAND
data = bernoulli.data.R
init = random initialization
init tries = 1
samples = samples.csv
append_samples = 0
save_warmup = 0
seed = 1845979644 (randomly generated)
chain_id = 1 (default)
iter = 2000
warmup = 1000
thin = 1 (default)
equal_step_sizes = 0
leapfrog_steps = -1
max_treedepth = 10
epsilon = -1
epsilon_pm = 0
delta = 0.5
gamma = 0.05
...

```

The ellipses (. . .) indicate that the output continues (as described below).

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

```

...
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 `samples.csv`.

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

```

...
# Samples Generated by Stan
#
# stan_version_major=1
# stan_version_minor=0
# stan_version_patch=0

```

```

# data=bernoulli.data.R
# init=random initialization
# append_samples=0
# save_warmup=0
# seed=1845979644
# chain_id=1
# iter=2000
# warmup=1000
# thin=1
# equal_step_sizes=0
# leapfrog_steps=-1
# max_treedepth=10
# epsilon=-1
# epsilon_pm=0
# delta=0.5
# gamma=0.05
...

```

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

```

...
lp__,treedepth__,stepsize__,theta
...

```

The first three names correspond to the log probability function value at the sample, the depth of tree evaluated by the NUTS sampler, and the step size. The single model parameter `theta` is stored in the fourth column.

Next up is the result of adaptation, reported as comments.

```

...
# step size=1.43707
# parameter step size multipliers:
# 1
...

```

This report says that NUTS step-size adaptation during warmup settled on a step size of 1.43707. The next two lines indicate the multipliers for scaling individual parameters, here just a single multiplier, 1, corresponding to the single parameter `theta`.

The rest of the file contains lines corresponding to the samples from each iteration.²

```

...
-7.07769,1,1.43707,0.158674

```

²There are repeated entries due to the Metropolis accept step in the no-U-turn sampling algorithm.

```

-7.07769,1,1.43707,0.158674
-7.37289,1,1.43707,0.130089
-7.09254,1,1.43707,0.361906
-7.09254,1,1.43707,0.361906
-7.09254,1,1.43707,0.361906
-6.96213,1,1.43707,0.337061
-6.96213,1,1.43707,0.337061
-6.77689,1,1.43707,0.220795
...
-6.85235,1,1.43707,0.195994
-6.85235,1,1.43707,0.195994
-6.81491,1,1.43707,0.20624
-6.81491,1,1.43707,0.20624
-6.81491,1,1.43707,0.20624
-6.81491,1,1.43707,0.20624
-6.81491,1,1.43707,0.20624

```

This ends the output.

Configuring Command-Line Options

The command-line options for running a model are detailed in the next chapter. They can also be printed on the command line in Linux and on the Mac as follows.

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

and on Windows with

```
> bernoulli --help
```

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 O=0 test-unit
```

That's the letter 'O' followed by an equal sign followed by the digit '0', which sets the tests to run at the lowest optimization level.

Warning: The `make` program may take 20+ minutes and consume 3+GB of memory to run the unit tests. Warnings can be safely ignored if the tests complete without a `FAIL` error.

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`).

¹As 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 -Ilib/boost_1.51.0 \
    -Ilib/eigen_3.1.0 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),
- `-Ilib/eigen_3.1.0` specifies the directory for the Eigen library,
- `-Ilib/boost_1.51.0` specifies the directory for the Boost 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 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.

4.1. Simple Example

Suppose the executable is in file `my_model` and the data is in file `my_data`, both in the current working directory. Then the Stan executable may be run on Windows using

```
> my_model --data=my_data
```

This will read the data from file `my_data`, run warmup tuning for 1000 iterations, which are discarded, then run the fully-adaptive NUTS sampler for 1000 iterations, writing the parameter (and other) values to the file `samples.csv` in the current working directory. A random number generation seed will be derived from the system time automatically. Note that on Mac or Linux, it is necessary to instead execute

```
> ./my_model --data=my_data
```

4.2. Parallel Example

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. To do so with four chains using a Bash shell on Mac or Linux execute

```
> for i in {1..4}
> do
>   ./my_model --refresh=0 --seed=12345 --chain_id=$i \
>     --samples=samples$i.csv --data=my_data &
> done
>
```

Note that there is blank line at the end that returns control to the prompt. The `&` at the end pushes each process into the background, so that the loop can continue without waiting for the previous chain to finish. This example requires several command-line options that are explained in further detail in the next subsection.

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

```
> for /l %x in (1, 1, 4) do start /b model --refresh=0 ^
>   --seed=12345 --chain_id=%x --samples=samples%x.csv
```

If the `grep` and `sed` programs are installed, then the following will combine the four csv files into a single csv file

```
> grep lp__ samples1.csv > combined.csv
> sed '/^[#1]/d' samples*.csv >> combined.csv
```

4.3. Command-Line Options

The executable Stan program is highly configurable. At the highest level, it may be configured for type of sampler (standard HMC or NUTS, with or without step size adaptation), it may be provided with data, initializations, a file to which to send the output, random number generation seeds, etc.

The full set of options is as follows. The next section provides some typical use-case examples.

`--help`

Display help information including the command call and options.
(default is not to display help information)

`--data=<file>`

Read data from specified dump formatted file
(required if model declares data)

`--init=<file>`

Use initial values from specified file or zero values if `<file>=0`
(default is random initialization)

`--samples=<file>`

File into which samples are written. Be sure to distinguish the file when executing multiple chains simultaneously.
(default = `samples.csv`)

`--append_samples`

Append samples to existing file if it exists; do not print headers
(default is to overwrite specified file)

`--seed=<int>`

Random number generation seed. When executing multiple chains simultaneously, it is best to give all of them the same seed but different values for `chain_id` (see below).
(default is to randomly generate from time)

`--chain_id=<int>`

Markov chain identifier, indexed from 1, to seed the L'Ecuyer random number generator to ensure non-overlapping random number sequences for each chain (?)
(default = 1)

```

--iter=<+int>
    Total number of iterations, including warmup
    (default = 2000)

--warmup=<+int>
    Discard the specified number of initial samples
    (default = iter / 2)

--thin=<+int>
    Period between saved samples after warm up
    (default = max(1, (iter - warmup) / 1000))

--refresh=<+int>
    Period between samples updating progress report print
    (default = max(1, iter / 200))

--leapfrog_steps=<int>
    Number of leapfrog steps; -1 for no-U-turn adaptation
    (default = -1)

--max_treedepth=<int>
    Limit NUTS leapfrog steps to pow(2, max_tree_depth); -1 for no limit
    (default = 10)

--epsilon=<float>
    Initial value for step size, or -1 to set automatically
    (default = -1)

--epsilon_pm=<[0, 1]>
    Sample epsilon uniformly from the interval
    [epsilon * (1 - epsilon_pm), epsilon * (1 + epsilon_pm)]
    (default = 0.0)

--equal_step_sizes
    Use unit mass matrix for NUTS; see related parameter epsilon
    (NUTS default is to estimate step sizes that vary by parameter)1

--delta=<+float>
    Initial step size for step-size adaptation
    (default = 0.5)

```

¹Stan's implementation of NUTS follows the approach in (? , section 5.4.2.4), which is equivalent to having different step sizes for each parameter. Varying step sizes are acceptable for HMC even though they vary the time scale of each parameter and thus depart from the Hamiltonian interpretation of the trajectory.

`--gamma=<+float>`

Gamma parameter for dual averaging step-size adaptation
(default = 0.05)

`--save_warmup`

Save the warmup samples as well as the post-warmup samples
(default is to not save the warmup samples)

`--test_grad`

Test gradient calculations using finite differences; if this option is chosen, only the gradient tests are done.
(default is not to test gradients)

4.4. Configuring the Sampler Type

Which sampler gets used, standard HMC or one of the two forms of NUTS, depends on the configuration of the command-line call. The following table summarizes.

<i>Sampler</i>	<i>Command Options</i>
HMC	<code>--epsilon=e</code> <code>--leapfrog_steps=L</code>
HMC + tuned step size	<code>--leapfrog_steps=L</code>
NUTS	<code>--epsilon=e</code>
NUTS + tuned step size	<code>--equal_step_sizes</code>
NUTS + tuned varying step sizes	(default configuration)

For an overview of Hamiltonian Monte Carlo (HMC), see (?). For details concerning the no-U-turn sampler (NUTS), see (??).

5. 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.

5.1. 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.

5.2. 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)
```

5.3. 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.

<code>y[1,1]</code>	<code>= 1</code>	<code>y[2,1]</code>	<code>= 3</code>	<code>y[3,1]</code>	<code>= 5</code>
<code>y[2,1]</code>	<code>= 2</code>	<code>y[2,2]</code>	<code>= 4</code>	<code>y[3,2]</code>	<code>= 6</code>

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 dimensions 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(c(1,2,3,4,5,6), .Dim = 2:3)
```

Similarly, the values may also be written with colon notation. Thus the following is equivalent to the last two variable definitions.

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

5.4. 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, 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 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.

Infinite and Not-a-Number Values

Stan's reader supports infinite and not-a-number values for scalar quantities (see Section 15.2.2 for more information). Both infinite and not-a-number values are supported by Stan's dump-format readers.

<i>Value</i>	<i>Preferred Form</i>	<i>Alternative Forms</i>
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.

5.5. 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)
```

5.6. 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`.

5.7. 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

6. Print Statements and Debugging

Before getting into specific statistical programming techniques in the following chapters, this chapter is dedicated to comments and print statements.

6.1. Comments

Stan supports C++-style comments; see Section 18.1 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);  
lp__ <- lp__ + y; // log Jacobian of exp(y) inverse transform
```

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 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!

6.2. Print Statements

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 the expression `y` along with the log probability accumulator, use the following statement.

```
print("y=", y, " log probability=", lp__);
```

This print statement prints the string “y=” followed by the value of the expression `y`, followed by the string “ log probability=” (with the leading space), followed by the value of the variable `lp__`.

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 17.7](#) for more information on the syntax and evaluation of print statements.

7. 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.

7.1. 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 observations¹ 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.

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.

¹A 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.

7.2. 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 parameter 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<lower=0> max_cov;  real<lower=0> min_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.

7.3. 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.

8. Truncated or Censored Data

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

8.1. 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 an informative prior in order for this model to not concentrate L strongly around $\min(y)$ and U strongly around $\max(y)$.

8.2. 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

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

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

$$\text{normal_cdf}(y, \mu, \sigma) = \Phi\left(\frac{y - \mu}{\sigma}\right).$$

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);
  lp__ <- lp__ + N_cens * log1m(normal_cdf(U, mu, sigma));
}
```

For the observed values in `y_obs`, the normal sampling model is used without truncation. The log probability accumulator `lp__` 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 for values of x near zero.

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

9. 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.

9.1. 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 ,

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

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

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

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.

9.2. 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

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

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

```
parameters {  
  real y;  
}  
model {
```

```

    lp__ <- 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

$$\begin{aligned}
 \log p_Y(y) &= \log(0.3 \times \text{Normal}(y|-1, 2) + 0.7 \times \text{Normal}(y|3, 1)) \\
 &= \log(\exp(\log(0.3 \times \text{Normal}(y|-1, 2))) \\
 &\quad + \exp(\log(0.7 \times \text{Normal}(y|3, 1)))) \\
 &= \log_sum_exp(\log(0.3) + \log \text{Normal}(y|-1, 2), \\
 &\quad \log(0.7) + \log \text{Normal}(y|3, 1)).
 \end{aligned}$$

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> 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);
    sigma[k] ~ uniform(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]);
    }
    lp__ <- lp__ + 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 pa-

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. 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.

10. Regression Models

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

10.1. 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

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

This is equivalent to the following sampling involving the residual,

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

and reducing still further, to

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

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

```
data {  
  int<lower=0> N;  
  real x[N];  
  real y[N];  
}  
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.

10.2. Coefficient and Noise Priors

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

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

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.¹

10.3. 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.

```
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.

¹Stan does not (yet) support truncated Cauchy distributions. The distributions which may be truncated are listed for discrete distributions in Chapter 24 and for continuous distributions in Chapter 25. Available truncated distributions may be found in the index by looking for suffix `_cdf`.

10.4. 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 respective 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 follows.

```
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

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

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]));
```

10.5. 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 $\text{Normal}(0, 5)$ priors on the coefficients is coded as follows.

```
data {
  int<lower=2> K;
  int<lower=0> N;
  int<lower=1> D;
  int<lower=1, upper=K> 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 $\gamma \in \mathbb{R}^K$ by

$$\text{softmax}(\gamma) = \left(\frac{\exp(\gamma_1)}{\sum_{k=1}^K \exp(\gamma_k)}, \dots, \frac{\exp(\gamma_K)}{\sum_{k=1}^K \exp(\gamma_k)} \right).$$

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

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 7.2 for an example of how to mix known quantities and unknown quantities in a vector.

10.6. Ordered Logistic and Probit Regression

Ordered regression for an outcome $y_n \in \{1, \dots, K\}$ 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 only needed 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]) - erf(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 $\text{Phi}(\eta - c[k])$ once and storing them for re-use.

10.7. 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, $l_n \in \{1, \dots, L\}$. 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> sigma[D];
    vector[D] beta[L];
}
model {
    for (d in 1:D) {
        mu[d] ~ normal(0,100);
        sigma[d] ~ uniform(0,1000);
        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]]));
}

```

10.8. 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 (2, pps. 314–320) for a textbook introduction to hierarchical IRT models and (2) 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
}

```

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 δ , which represents the average student's response to the average question.² 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)
}
```

This model uses the logit-parameterized Bernoulli distribution, where

$$\text{bernoulli_logit}(y|\alpha) = \text{bernoulli}(y|\text{logit}^{-1}(\alpha)).$$

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

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

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

²(?) treat the δ term equivalently as the location parameter in the distribution of student abilities.

and β located at 0 identifies the parameters; see (?) 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 IPL 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 **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.

10.9. 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 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);  
}  
}
```

11. Reparameterizations and Changes 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.

11.1. 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. from 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 (2, 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 parameter 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));
  ...
}

```

Univariate transforms must be monotonic and differentiable everywhere in their support. Stan takes care of the derivative involved in the gradient behind the scenes.

Multivariate transforms are more complex. They require the log absolute Jacobian to be applied. In Stan, this requires coding the transform and the Jacobian. In the full case (where the Jacobian is not triangular), the code would look as follows.

```
parameters {  
  
}
```

11.2. Changes of Variables

Changes of variables are applied when the transformation of a parameter has a specified distribution. One textbook example is the lognormal distribution, which is the distribution of a variable $y > 0$ whose logarithm $\log y$ has a normal distribution.

Univariate Changes of Variables

The change of variables requires an adjustment to the probability to account for the distortion caused by the transform. In the case of univariate distributions, the resulting probability must be scaled by the absolute derivative of the transform (see Section 28.1.1 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

$$p(y) = \text{Normal}(\log y | \mu, \sigma) \left| \frac{d}{dy} \log y \right| = \text{Normal}(\log y | \mu, \sigma) \frac{1}{y}.$$

Stan works on the log scale to prevent underflow, where

$$\log p(y) = \log \text{Normal}(\log y | \mu, \sigma) - \log y.$$

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);  
  lp__ <- lp__ - 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.

¹This 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 25.3.1).

```

model {
  real log_y;
  log_y <- log(y);
  log_y ~ normal(mu, sigma);
  lp__ <- lp__ - 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.

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 Section 28.1.2 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] ...
  lp__ <- lp__ + 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 $u[1], \dots, u[k]$ 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] ...
lp__ <- lp__ + sum(log(J_diag));
...
```

12. 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 log probability accumulator `lp__` with the log probability. The rest of the chapter provides two examples.

12.1. 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.

$$\text{Triangle}(y|\alpha, \beta) = \frac{2}{\beta - \alpha} \left(1 - \left| y - \frac{\alpha + \beta}{2} \right| \right)$$

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 `Triangle(-1, 1)`.

```
parameters {  
  real<lower=-1, upper=1> y;  
}  
model {  
  lp__ <- lp__ + log1m(fabs(y));  
}
```

The single scalar parameter `y` is declared as lying in the interval $(-1, 1)$. The log probability variable `lp__` 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 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 log probability variable `lp__` is incremented in this program rather than being set. This is because the transform involved for the bounded variable `y` of type `real<lower=-1, upper=1>` implicitly adds a term to `lp__` to adjust the log probability for the transform (adding the log absolute derivative of the inverse transform, a scaled inverse logit).

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 $(-1, 1)$.

```
lp__ <- 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.

```
lp__ <- lp__ + 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).

¹The 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 28 for more information on the transforms used by Stan.

12.2. Poisson Cumulative Distribution Function

As of the original writing of this section, Stan does not have a complete set of cumulative distribution functions. For example, the cumulative Poisson distribution is missing.² Thus in order to write a truncated Poisson distribution, its cumulative distribution must be coded by hand. The cumulative distribution for the Poisson with rate $\lambda \in (0, \infty)$ at outcome $n \in \mathbb{N}$ is

$$\frac{\Gamma(n+1, \lambda)}{n!}.$$

The function $\Gamma()$ here is the incomplete gamma function, defined recursively by the base case

$$\Gamma(\Gamma(n, \lambda) = (n-1)\Gamma(n-1, \lambda) + \lambda^{n-1} \exp(-\lambda)$$

²The distributions for which cumulative distribution functions are available can be found by browsing the index looking for the suffix `_cdf`.

13. Optimizing Stan Code

This chapter provides a grab bag of techniques for optimizing Stan code, including vectorization, sufficient statistics, and conjugacy.

13.1. 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

(?) 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

$$p(y, x) = \text{Normal}(y|0, 3) \times \prod_{n=1}^9 \text{Normal}(x_n|0, \exp(y/2)).$$

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 13.1.

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));  
}
```

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.

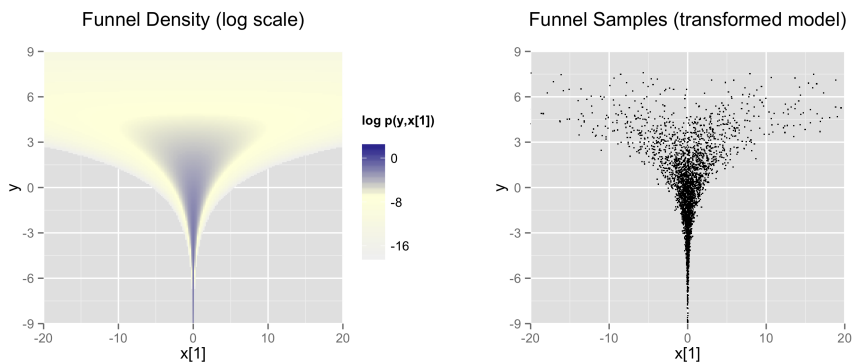


Figure 13.1: *Neal’s Funnel.* (Left) The marginal density of Neal’s funnel for the upper-level variable y and one lower-level variable x_1 (see the text for the formula). The blue region has log density greater than -8 , the yellow region density greater than -16 , and the gray background a density less 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.

```
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.

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 μ and Σ are fixed data, but they could be unknown parameters, in which case their priors would be unaffected by a reparameterization of β .

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 α and second, it avoids the need to invert Σ 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 τ were fixed data but the vector of standard deviations σ were unknown parameters, then a reparameterization of β in terms of α 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 $W = L A A^\top L^\top$ is distributed Wishart with scale matrix $S = L L^\top$ 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) {
    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 $W = L * A * A^T * L^T$, $W^{-1} = L^{T^{-1}} A^{T^{-1}} A^{-1} L^{-1}$, where all four inverses exist, but $L^{-1^T} = L^{T^{-1}}$ and $A^{-1^T} = A^{T^{-1}}$. 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) {
    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. ? 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[K] z[beta];
  ...
transformed parameters {
  simplex[K] rho;
  for (k in 1:K)
    rho[k] <- dot_self(z[k]); // sum-of-squares
  rho <- rho / sum(rho);
}
model {
  z ~ normal(0,1);
  // implies: rho ~ dirichlet(0.5 * beta * ones)
  ...

```

13.2. 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 allocated¹ 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.

¹Stan uses its own arena-based allocation, so allocation and deallocation are faster than with a raw call to `new`.

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.

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.² 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';
}

```

²Future 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.

```

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.

```

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 25.1 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.

13.3. 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 θ . 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 $\text{sum}(y)$ of a binary vector y is equal to the number of positive outcomes out of a total of N trials.

13.4. 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 $\text{sum}(y)$ could be used in the place of $\text{sum}(y)$.

Part IV

Modeling Language Reference

14. Execution of a Stan Program

This chapter provides a sketch of how a compiled Stan model is executed. This sketch is elaborated in the following chapters of this part, which cover variable declarations, expressions, statements, and blocks in more detail.

14.1. Before Sampling

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 5).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.

Initialization

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.

If there are no user-supplied initial values, the unconstrained initial values are generated uniformly from the interval $(-2, 2)$.

¹The 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.

14.2. 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 (?).

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.

14.3. 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.

14.4. 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

²In the R version of Stan, the values may either be written to a csv file or directly back to R's memory.

15. 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.

15.1. 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 three constrained vector data types, `simplex` for unit simplexes and `ordered` for ordered vectors of scalars and `positive_ordered` for vectors of positive ordered scalars. There are two specialized matrix data types, `corr_matrix` and `cov_matrix`, for correlation matrices (symmetric, positive definite, unit diagonal) and covariance matrices (symmetric, positive definite).

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.

15.2. 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 20.1 for more information).

Reals

Stan uses 64-bit (8-byte) floating point representations of real numbers. Stan roughly¹ 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.”

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.

¹Stan 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.

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.

15.3. 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=fmin(y), upper=fmax(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.

15.4. 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.

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 10.6).

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;
```

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 $K \times K$ correlation matrix, where K is the value of the data variable `K`.

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]` 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

²As of Stan version 1.0, the form `m[2,3]` is more efficient because it does not require the creation and use of

15.5. 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

```
cov_matrix[5] mu[2,3,4];
```

of a $2 \times 3 \times 4$ array of 5×5 covariance matrices.

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.

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]
```

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[4,3];
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[4,3,5,2];     // 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`.

15.6. Types versus Sizes

The size associated with a given variable is not part of its data type. Sizes are determined dynamically (at run time) and thus cannot be type-checked statically.

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.

16. 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.

16.1. 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 15.2.1).

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} .

16.2. 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.

The following list contains reserved words for Stan's programming language. Not all of these features are implemented, 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, ordered, positive_ordered,  
row_vector, matrix, corr_matrix, cov_matrix.
```

Variable names will not conflict with the following block identifiers,

```
model, data, parameters, quantities, transformed,  
generated,
```

nor will they conflict with the names of predefined functions or distributions. Nevertheless, these names should be avoided for the sake of program clarity.

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.

16.3. 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.

16.4. 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 23.

For example, if y and μ are variables of type `vector` and Σ 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.

$$(((y - \mu)') * \text{Sigma}) * (y - \mu)$$

The type of subexpression $(y - \mu)' * \text{Sigma}$ 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 16.1. 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 $(u')*v$.

16.5. 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`.

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.

<i>Op.</i>	<i>Prec.</i>	<i>Assoc.</i>	<i>Placement</i>	<i>Description</i>
	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 16.1: *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 binded 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 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`.

16.6. 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 21.1 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

<i>Type</i>	<i>Primitive Type</i>	<i>Type</i>	<i>Primitive Type</i>
<code>int</code>	<code>int</code>	<code>vector</code>	<code>vector</code>
<code>real</code>	<code>real</code>	<code>simplex</code>	<code>vector</code>
<code>matrix</code>	<code>matrix</code>	<code>ordered</code>	<code>vector</code>
<code>cov_matrix</code>	<code>matrix</code>	<code>pos_ordered</code>	<code>vector</code>
<code>corr_matrix</code>	<code>matrix</code>	<code>row_vector</code>	<code>row_vector</code>

Figure 16.2: 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.

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.

16.7. 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 16.2 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`.

¹Languages 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 e_1, \dots, e_N are expressions for $N \geq 0$, then $f(e_1, \dots, e_N)$ is an expression whose type is determined by the return type in the function signature for f given e_1 through e_N . 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 Section 16.6.3 for an exact specification of Stan's integer-to-real type-promotion rule).

In general, `matirx` 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)`.

17. Statements

The blocks of a Stan program (see Chapter 18) 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.

17.1. 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 1.0, 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 1.0 does not support providing an array of indices as an argument to create a piecemeal subarray of a larger array.

17.2. 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 written as an assignment statement using the reserved variable `lp__` as

```
lp__ <- lp__ + normal_log(y,mu,sigma);
```

The variable `lp__` acts as an accumulator for the log (proportional) probability defined by the model as a function of the parameters and data.

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,

```
lp__ <- lp__ + 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`.

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 28.1 for full definitions). For the case above, the following adjustment will account for the log transform.¹

```
lp__ <- lp__ - log(fabs(y));
```

¹Because $\log \left| \frac{d}{dy} \log y \right| = \log |1/y| = -\log |y|$; see Section 28.1.

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

$$p_{(a,b)}(x) = \frac{p(x)}{\int_a^b p(x') dx'}.$$

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) as

```
y ~ normal(0,1);
lp__ <- lp__ - 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

$$\int_{-\infty}^{2.1} \text{Normal}(x|0,1) dx,$$

which is the probability a unit normal variable takes on values less than 2.1, or about 0.98.

As with constrained variable declarations, truncation can be one sided. The density $p(x)$ 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 represented as

```
y ~ normal(0,1) T[-0.5,];
```

The truncation has the same effect as the following direct update to the accumulated log probability.

```
y ~ normal(0,1);
lp__ <- lp__ - log(1 - normal_cdf(-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

$$p_{(,b)}(x) = \frac{p(x)}{\int_{-\infty}^b p(x') dx'}.$$

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.

```
y ~ normal(0,1);  
lp__ <- lp__ - log(normal_cdf(2.1,0,1));
```

In all cases, the truncation is only well formed if there is an appropriate cumulative distribution function defined.² Chapter 24 and Chapter 25 document the available discrete and continuous cumulative distribution 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 log probability accumulator `lp__`, which in turn results in the sample being rejected; see Section 8.1.1 for programming strategies to keep all values within bounds.

Scope of `lp__`

The variable `lp__` is only available in the `parameter`, `transformed parameter`, and `model blocks` (see Chapter 18). The variable `lp__` is undefined (and may not be declared by the user) in the `data`, `transformed data` and `generated quantity blocks`.

17.3. 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.

²Some cumulative distributions and their gradients present computational challenges because they lack simple, analytic forms. More cumulative distributions will be added in future releases.

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.³

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.

```
total <- 0.0;
for (n in 1:N)
  total <- total + x[n];
```

³A 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 .

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;  
}
```

17.4. 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.

17.5. 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 true (non-zero) value, the body statement is executed, then an attempt is made to execute the entire while-loop again. If the condition evaluates to a false (zero) value

while-loop statement completes. If the condition evaluates to a non-zero value, the statement is executed and control, then the loop is evaluated again, starting with the condition. Thus the while loop executes the statement for as many times as the condition evaluates to true.

17.6. 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.

17.7. 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.⁴ 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]
```

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.

⁴The adjoint component is always zero during execution for the algorithmic differentiation variables used to implement parameters, transformed parameters, and local variables in the model.

```

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 or infinite values, both of which will be printed.

18. 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.

18.1. 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.

18.2. 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 ...  
}  
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.¹ 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

¹If the C++ code is configured for concurrent threads, the data and transformed data blocks can be executed once and reused for multiple chains.

<i>Block</i>	<i>Stmt</i>	<i>Action / Period</i>
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
<i>(initialization)</i>	n/a	read, transform / chain

Figure 18.1: 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.

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 18.1. Another way to look at the variables is in terms of their function. To decide which variable to use, consult the charts in Figure 18.2. The last line has no corresponding location, as there is no need to print a variable every iteration that does not depend on parameters.² The rest of this chapter provides full details on when and how the variables and statements in each block are executed.

²It 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.

<i>Params</i>	<i>Log Prob</i>	<i>Print</i>	<i>Declare In</i>
+	+	+	transformed parameters
+	+	–	<i>local in</i> model
+	–	–	<i>local in</i> generated quantities
+	–	+	generated quantities
–	–	+	generated quantities*
–	±	–	<i>local in</i> transformed data
–	+	+	transformed data <i>and</i> generated quantities*

Figure 18.2: *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 2 in this chapter).*

18.3. Statistical Variable Taxonomy

(?, p. 366) provides a taxonomy of the kinds of variables used in Bayesian models. Figure 18.3 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. 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;
}

```

<i>Variable Kind</i>	<i>Declaration Block</i>
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 18.3: Variables of the kind indicated in the left column must be declared in one of the blocks declared in the right column.

```

parameters {
  real mu_y;                // modeled param
  real<lower=0> tau_y;       // modeled param
}
transformed parameters {
  real<lower=0> sigma_y;     // derived quantity (param)
  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.

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.

18.4. 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.³

Data variables are not transformed in any way. The format for data files is provided in Chapter 5.

³With 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 `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.

18.5. 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.

Statements

The statements in a `transformed data` block are used to define (provide values for) variables declared in the `transformed data` block. The special variable `lp_` is not defined for the statements in the `transformed data` block. Consequently, probability statements, which implicitly access `lp_`, may not be used. Furthermore, 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 3 in this chapter).

Variables declared in the `data` block may be used in statements in the `transformed data` block.

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.

18.6. 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 28.

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.

18.7. 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.

18.8. 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.

18.9. 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 26 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.

19. 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.

19.1. 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 ']'
           | 'row_vector' range_constraint '[' expression ']'
           | 'matrix' range_constraint '[' 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 ']'

```

19.2. 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 [16.4.1](#), 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

20. Integer-Valued Basic Functions

This chapter describes Stan's built-in function that take various types of arguments and return results of type integer.

20.1. 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.

20.2. 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

20.3. 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

21. Real-Valued Basic Functions

This chapter describes built-in functions that take zero or more real or integer arguments and return real values.

21.1. 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

21.2. 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 **epsilon**()

The smallest positive real value representable

real **negative_epsilon**()

The largest negative real value representable

21.3. 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 16.1.

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 $\neg a \& \& \neg b$ is interpreted as $(\neg a) \& \& \neg b$, and $a \< \neg b \mid \mid \neg c \> \neg d \& \& \neg e \neg f$ as $(a \< \neg b) \mid \mid \neg ((\neg c \> \neg d) \& \& \neg (e \neg f))$.

```

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

```
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)
    0 if x is negative and 1 otherwise; equivalent to x > 0.0
```

21.4. 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.

21.5. Absolute Functions

real **abs**(real x)

The absolute value of x

real **fabs**(real x)

The absolute value of x

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

21.6. Bounds Functions

real **fmin**(real x , real y)

The minimum of x and y

real **fmax**(real x , real y)

The maximum of x and y

21.7. Arithmetic Functions

real **fmod**(real x , real y)

The real value remainder after dividing x by y

21.8. Rounding Functions

real **floor**(real x)

The floor of x , which is the largest integer less than or equal to x , converted to a real value

real **ceil**(real x)

The ceiling of x , which is the smallest integer greater than or equal to x , converted to a real value

real **round**(real x)

The nearest integer to x , converted to a real value

real **trunc**(real x)

The integer nearest to but no larger in magnitude than x , converted to a double value

21.9. Power and Logarithm Functions

real **sqrt**(real x)

The square root of x

real **cbirt**(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

21.10. 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

21.11. 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

21.12. Link Functions

The following functions are commonly used as link functions in generalized linear models (see Section 10.4).

`real logit(real x)`

The log odds, or logit, function applied to x

`real inv.logit(real x)`

The logistic sigmoid function applied to x

`real inv.cloglog(real x)`

The inverse of the complement log-log function applied to x

21.13. 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

`real binary.log.loss(int y , real y_hat)`

The log loss of predicting probability y_hat for binary outcome y

21.14. Combinatorial Functions

`real tgamma(real x)`

The gamma function applied to x

`real lgamma(real x)`

The natural logarithm of the gamma function applied to x

`real lmgamma(int n , real x)`

The natural logarithm of the multinomial gamma function with n dimensions applied to x

```
real lbeta(real x, real y)
```

The natural logarithm of the beta function applied to x

```
real binomial_coefficient_log(real x, real y)
```

The natural logarithm of the binomial coefficient of x choose y , generalized to real values via the gamma function

21.15. 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

```
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 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

22. Array Operations

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

real **sum**(real $x[]$)

The sum of the elements in x , or 0 if x is empty.

real **sum**(int $x[]$)

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 **mean**(real $x[]$)

The sample mean of the elements in x , or raise exception if x is empty

real **variance**(real $x[]$)

The sample variance of the elements in x (based on dividing by $\text{length} - 1$), or 0 if x is empty

real **sd**(real $x[]$)

The sample standard deviation of elements in x (divide by $\text{length} - 1$), or 0 if x is empty

real **log_sum_exp**(real $x[]$)

The natural logarithm of the sum of the exponentials of the elements in x

23. Matrix Operations

23.1. 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

23.2. 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

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, \dots, x_N is the sequence y_1, \dots, y_N , where

$$y_n = \sum_{m=1}^n x_m.$$

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

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

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 xx^T .

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 $x^T x$.

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 $n \leq m$ and $L(m, n)$ equal to 0 if $n > m$, the result is the symmetric matrix LL^T . 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 formed from the the row vector rv , i.e., `m * diag_matrix(rv)`.

23.3. Reductions

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

real **mean**(vector x)

The sample mean of the values in x , or raise exception if x is empty

real **mean**(row_vector x)

The sample mean of the values in x , or raise exception if x is empty

real **mean**(matrix x)

The sample mean of the values in x , or raise exception if x is empty

real **variance**(vector x)

The sample variance of the values in x (divide by size minus 1), or 0 if x is empty

real **variance**(row_vector x)

The sample variance of the values in x (divide by size minus 1), or 0 if x is empty

real **variance**(matrix x)

The sample variance of the values in x (divide by size minus 1), or 0 if x is empty

real **sd**(vector x)

The sample standard deviation of the values in x (divide by size minus 1), or 0 if x is empty

real **sd**(row_vector x)

The sample standard deviation of the values in x (divide by size minus 1), or 0 if x is empty

real **sd**(matrix x)

The sample standard deviation of the values in x (divide by size minus 1), or 0 if x is empty

23.4. 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

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

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.

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'*

23.5. Special Matrix Functions

The softmax function maps \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*.

`vector softmax(vector x)`

The softmax of *x*

23.6. 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)`

`vector operator\(matrix A, vector b)`

The 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 $\text{tri}(A)$ is defined by

$$\text{tri}(A)[m, n] = \begin{cases} A[m, n] & \text{if } m \geq n, \text{ and} \\ 0 & \text{otherwise.} \end{cases}$$

row_vector **mdivide_right_tri_low**(row_vector b , matrix a)

The right division of b by $\text{tri}(a)$, a lower triangular view of a ; equivalently $b * \text{inverse}(\text{tri}(a))$

matrix **mdivide_right_tri_low**(matrix b , matrix a)

The right division of b by $\text{tri}(a)$, a lower triangular view of a ; equivalently $b * \text{inverse}(\text{tri}(a))$

vector **mdivide_left_tri_low**(matrix a , vector b)

The left division of b by a triangular view of $\text{tri}(a)$, a lower triangular view of a ; equivalently $\text{inverse}(\text{tri}(a)) * b$

matrix **mdivide_left_tri_low**(matrix a , matrix b)

The left division of b by a triangular view of $\text{tri}(a)$, a lower triangular view of a ; equivalently $\text{inverse}(\text{tri}(a)) * b$

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

matrix **inverse**(matrix A)

The inverse of A

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

24. Discrete Probabilities

The various discrete probability functions are organized by their support.

24.1. Binary Discrete Probabilities

Binary probability functions have support on $\{0, 1\}$, where 1 represents the value true and 0 the value false.

Bernoulli Distribution

If $\theta \in [0, 1]$, then for $y \in \{0, 1\}$,

$$\text{Bernoulli}(y|\theta) = \begin{cases} \theta & \text{if } y = 1, \text{ and} \\ 1 - \theta & \text{if } y = 0. \end{cases}$$

real **bernoulli_log**(ints *y*, reals *theta*)

The log Bernoulli probability mass of *y* given chance of success *theta*

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.

If $\alpha \in \mathbb{R}$, then for $c \in \{0, 1\}$,

$$\text{BernoulliLogit}(c|\alpha) = \text{Bernoulli}(c|\text{logit}^{-1}(\alpha)) = \begin{cases} \text{logit}^{-1}(\alpha) & \text{if } y = 1, \text{ and} \\ 1 - \text{logit}^{-1}(\alpha) & \text{if } y = 0. \end{cases}$$

real **bernoulli_logit_log**(ints *y*, reals *alpha*)

The log Bernoulli probability mass of *y* given logit chance of success *exp(alpha)*

24.2. Bounded Discrete Probabilities

Bounded discrete probability functions have support on $\{0, \dots, N\}$ for some upper bound N .

Binomial Distribution

If $N \in \mathbb{N}$ and $\theta \in [0, 1]$, then for $n \in \{0, \dots, N\}$,

$$\text{Binomial}(n|N, \theta) = \binom{N}{n} \theta^n (1 - \theta)^{N-n}.$$

real **binomial.log**(int n , int N , real θ)

The log binomial probability mass of n successes in N trials given chance of success θ

Beta-Binomial Distribution

If $N \in \mathbb{N}$, $\alpha \in \mathbb{R}^+$, and $\beta \in \mathbb{R}^+$, then for $n \in \{0, \dots, N\}$,

$$\text{BetaBinomial}(n|N, \alpha, \beta) = \binom{N}{n} \frac{B(n + \alpha, N - n + \beta)}{B(\alpha, \beta)},$$

where the beta function $B(x, y)$ is defined for $x \in \mathbb{R}^+$ and $y \in \mathbb{R}^+$ by

$$B(x, y) = \frac{\Gamma(x) \Gamma(y)}{\Gamma(x + y)}.$$

real **beta_binomial.log**(ints n , ints N , reals α , reals β)

The log beta-binomial probability mass of n successes in N trials given prior success count (plus one) of α and prior failure count (plus one) of β

Hypergeometric Distribution

If $a \in \mathbb{N}$, $b \in \mathbb{N}$, and $N \in \{0, \dots, a + b\}$, then for $n \in \{0, \dots, \min(a, N)\}$,

$$\text{Hypergeometric}(n|N, a, b) = \frac{\binom{a}{n} \binom{b}{N-n}}{\binom{a+b}{N}}.$$

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

Categorical Distribution

If $N \in \mathbb{N}$ and $\theta \in N$ -simplex, then for $y \in \{1, \dots, N\}$,

$$\text{Categorical}(y|\theta) = \theta_n.$$

real **categorical.log**(int y , vector θ)

The log categorical probability mass function with outcome y in $1 : N$ given N -simplex distribution parameter θ

Ordered Logistic Distribution

If $K \in \mathbb{N}$ with $K > 2$, $c \in \mathbb{R}^{K-1}$ such that $c_k < c_{k+1}$ for $k \in \{1, \dots, K-2\}$, and $\eta \in \mathbb{R}$, then for $k \in \{1, \dots, K\}$,

$$\text{OrderedLogistic}(k|\eta, c) = \begin{cases} 1 - \text{logit}^{-1}(\eta - c_1) & \text{if } k = 1, \\ \text{logit}^{-1}(\eta - c_{k-1}) - \text{logit}^{-1}(\eta - c_k) & \text{if } 1 < k < K, \text{ and} \\ \text{logit}^{-1}(\eta - c_{K-1}) - 0 & \text{if } k = K. \end{cases}$$

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 $\text{logit}^{-1}(\infty) = 1$.

real **ordered_logistic_log**(int *k*, real *eta*, vector *c*)

The log ordered logistic probability mass of k given linear predictor *eta* and cut-points *c*.

24.3. Unbounded Discrete Distributions

The unbounded discrete distributions have support over the natural numbers (i.e., the non-negative integers).

Negative Binomial Distribution

If $\alpha \in \mathbb{R}^+$ and $\beta \in \mathbb{R}^+$, then for $n \in \mathbb{N}$,

$$\text{NegativeBinomial}(n|\alpha, \beta) = \binom{n + \alpha - 1}{\alpha - 1} \left(\frac{\beta}{\beta + 1} \right)^\alpha \left(\frac{1}{\beta + 1} \right)^n.$$

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*

Poisson Distribution

If $\lambda \in \mathbb{R}^+$, then for $n \in \mathbb{N}$,

$$\text{Poisson}(n|\lambda) = \frac{1}{n!} \lambda^n \exp(-\lambda).$$

real **poisson_log**(ints *n*, reals *lambda*)

The log Poisson probability mass of n given rate *lambda*

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.

If $\alpha \in \mathbb{R}$, then for $n \in \mathbb{N}$,

$$\text{PoissonLog}(n|\alpha) = \frac{1}{n!} \frac{1}{\alpha} (\log \alpha)^n.$$

real **poisson_log_log**(ints *n*, reals *alpha*)

The log Poisson probability mass of *n* given log rate *lambda*

24.4. Multivariate Discrete Probabilities

Multinomial Distribution

If $K \in \mathbb{N}$, $N \in \mathbb{N}$, and $\theta \in K$ -simplex, then for $y \in \mathbb{N}^K$ such that $\sum_{k=1}^K y_k = N$,

$$\text{Multinomial}(y|\theta, N) = \binom{N}{y_1, \dots, y_K} \prod_{k=1}^K \theta_k^{y_k},$$

where the multinomial coefficient is defined by

$$\binom{N}{y_1, \dots, y_K} = \frac{N!}{\prod_{k=1}^K y_k!}.$$

real **multinomial_log**(int[] *y*, vector *theta*)

The log multinomial probability mass function with outcome array *y* of size *K* given the *K*-simplex distribution parameter *theta* and (implicit) total count $N = \text{sum}(y)$

25. Continuous Probabilities

The various continuous probability functions are organized by their support.

25.1. Vectorization

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.

25.2. Univariate Continuous Probabilities

The univariate continuous probability distributions have support on all real numbers.

Normal Distribution

If $\mu \in \mathbb{R}$ and $\sigma \in \mathbb{R}^+$, then for $y \in \mathbb{R}$,

$$\text{Normal}(y|\mu, \sigma) = \frac{1}{\sqrt{2\pi} \sigma} \exp\left(-\frac{1}{2} \left(\frac{\theta - \mu}{\sigma}\right)^2\right).$$

```
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 $sigma$

Student- t Distribution

If $\nu \in \mathbb{R}^+$, $\mu \in \mathbb{R}$, and $\sigma \in \mathbb{R}^+$, then for $y \in \mathbb{R}$,

$$\text{StudentT}(y|\nu, \mu, \sigma) = \frac{\Gamma((\nu+1)/2)}{\Gamma(\nu/2)} \frac{1}{\sqrt{\nu\pi} \sigma} \left(1 + \frac{1}{\nu} \left(\frac{\theta - \mu}{\sigma}\right)^2\right)^{-(\nu+1)/2}.$$

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 $sigma$

Cauchy Distribution

If $\mu \in \mathbb{R}$ and $\sigma \in \mathbb{R}^+$, then for $y \in \mathbb{R}$,

$$\text{Cauchy}(y|\mu, \sigma) = \frac{1}{\pi\sigma} \frac{1}{1 + ((y - \mu)/\sigma)^2}.$$

real **cauchy_log**(reals y , reals mu , reals $sigma$)

The log of the Cauchy density of y given location mu and scale $sigma$

Double Exponential (Laplace) Distribution

If $\mu \in \mathbb{R}$ and $\sigma \in \mathbb{R}^+$, then for $y \in \mathbb{R}$,

$$\text{DoubleExponential}(y|\mu, \sigma) = \frac{1}{2\sigma} \exp\left(-\frac{|y - \mu|}{\sigma}\right).$$

real **double_exponential_log**(reals y , reals mu , reals $sigma$)

The log of the double exponential density of y given location mu and scale $sigma$

Logistic Distribution

If $\mu \in \mathbb{R}$ and $\sigma \in \mathbb{R}^+$, then for $y \in \mathbb{R}$,

$$\text{Logistic}(y|\mu, \sigma) = \frac{1}{\sigma} \exp\left(-\frac{y - \mu}{\sigma}\right) \left(1 + \exp\left(-\frac{y - \mu}{\sigma}\right)\right)^{-2}.$$

real **logistic_log**(reals y , reals mu , reals $sigma$)

The log of the logistic density of y given location mu and scale $sigma$

25.3. Positive Continuous Probabilities

The positive continuous probability functions have support on the positive real numbers.

Log-Normal Distribution

If $\mu \in \mathbb{R}$ and $\sigma \in \mathbb{R}^+$, then for $y \in \mathbb{R}^+$,

$$\text{LogNormal}(y|\mu, \sigma) = \frac{1}{\sqrt{2\pi}} \frac{1}{\sigma} \frac{1}{y} \exp\left(-\frac{1}{2} \left(\frac{\log y - \mu}{\sigma}\right)^2\right).$$

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**(real y , real mu , real $sigma$)

The cumulative lognormal distribution of y given location mu and scale $sigma$

Chi-Square Distribution

If $\nu \in \mathbb{R}^+$, then for $y \in \mathbb{R}^+$,

$$\text{ChiSquare}(y|\nu) = \frac{2^{-\nu/2}}{\Gamma(\nu/2)} y^{\nu/2-1} \exp\left(-\frac{1}{2} y\right).$$

real **chi.square.log**(reals y , reals nu)

The log of the Chi-square density of y given degrees of freedom nu

Inverse Chi-Square Distribution

If $\nu \in \mathbb{R}^+$, then for $y \in \mathbb{R}^+$,

$$\text{InvChiSquare}(y|\nu) = \frac{2^{-\nu/2}}{\Gamma(\nu/2)} y^{-(\nu/2-1)} \exp\left(-\frac{1}{2} \frac{1}{y}\right).$$

real **inv.chi.square.log**(reals y , reals nu)

The log of the inverse Chi-square density of y given degrees of freedom nu

Scaled Inverse Chi-Square Distribution

If $\nu \in \mathbb{R}^+$ and $\sigma \in \mathbb{R}^+$, then for $y \in \mathbb{R}^+$,

$$\text{ScaledInvChiSquare}(y|\nu) = \frac{(\nu/2)^{\nu/2}}{\Gamma(\nu/2)} \sigma^\nu y^{-(\nu/2+1)} \exp\left(-\frac{1}{2} \nu \sigma^2 \frac{1}{y}\right).$$

real **scaled.inv.chi.square.log**(reals y , reals nu , reals s)

The log of the scaled inverse Chi-square density of y given degrees of freedom nu and scale s

Exponential Distribution

If $\beta \in \mathbb{R}^+$, then for $y \in \mathbb{R}^+$,

$$\text{Exponential}(y|\beta) = \beta \exp(-\beta y).$$

`real exponential.log(reals y, reals beta)`

The log of the exponential density of y given inverse scale *beta*

`real exponential.cdf(real y, real beta)`

The cumulative distribution function of y given inverse scale *beta*

Gamma Distribution

If $\alpha \in \mathbb{R}^+$ and $\beta \in \mathbb{R}^+$, then for $y \in \mathbb{R}^+$,

$$\text{Gamma}(y|\alpha, \beta) = \frac{\beta^\alpha}{\Gamma(\alpha)} y^{\alpha-1} \exp(-\beta y).$$

`real gamma.log(reals y, reals alpha, reals beta)`

The log of the gamma density of y given shape *alpha* and inverse scale *beta*

Inverse Gamma Distribution

If $\alpha \in \mathbb{R}^+$ and $\beta \in \mathbb{R}^+$, then for $y \in \mathbb{R}^+$,

$$\text{InvGamma}(y|\alpha, \beta) = \frac{\beta^\alpha}{\Gamma(\alpha)} y^{-(\alpha+1)} \exp\left(-\beta \frac{1}{y}\right).$$

`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*

Weibull Distribution

If $\alpha \in \mathbb{R}^+$ and $\sigma \in [0, \infty)$, then for $y \in \mathbb{R}^+$,

$$\text{Weibull}(y|\alpha, \sigma) = \frac{\alpha}{\sigma} \left(\frac{y}{\sigma}\right)^{\alpha-1} \exp\left(-\left(\frac{y}{\sigma}\right)^\alpha\right).$$

`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(real y, real alpha, real sigma)`

The Weibull cumulative distribution of y given shape *alpha* and scale *sigma*

25.4. Positive Lower-Bounded Probabilities

The positive lower-bounded probabilities have support on real values above some positive minimum value.

Pareto Distribution

If $y_0 \in \mathbb{R}^+$ and $\alpha \in \mathbb{R}^+$, then for $y \in \mathbb{R}^+$,

$$\text{Pareto}(y|y_0, \alpha) = \alpha y_0 \left(\frac{1}{y} \right)^{\alpha+1}.$$

`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 $alpha$

25.5. $[0, 1]$ Continuous Probabilities

Beta Distribution

If $\alpha \in \mathbb{R}^+$ and $\beta \in \mathbb{R}^+$, then for $\theta \in [0, 1]$,

$$\text{Beta}(\theta|\alpha, \beta) = \frac{1}{B(\alpha, \beta)} \theta^{\alpha-1} (1 - \theta)^{\beta-1},$$

where the beta function $B()$ is as defined in Section 24.2

`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) $beta$

25.6. Bounded Continuous Probabilities

The bounded continuous probabilities have support on a finite interval of real numbers.

Uniform Distribution

If $\alpha \in \mathbb{R}$ and $\beta \in (\alpha, \infty)$, then for $y \in [\alpha, \beta]$,

$$\text{Uniform}(y|\alpha, \beta) = \frac{1}{\beta - \alpha}.$$

`real uniform.log(reals y, reals alpha, reals beta)`

The log of the uniform density of y given lower bound $alpha$ and upper bound $beta$

25.7. Simplex Probabilities

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 $k \in \{1, \dots, K\}$ and $\sum_{k=1}^K \theta_k = 1$.

Dirichlet Distribution

If $K \in \mathbb{N}$ and $\alpha \in (\mathbb{R}^+)^K$, then for $\theta \in K$ -simplex,

$$\text{Dirichlet}(\theta|\alpha) = \frac{\Gamma\left(\sum_{k=1}^K \alpha_k\right)}{\prod_{k=1}^K \Gamma(\alpha_k)} \prod_{k=1}^K \theta_k^{\alpha_k-1}.$$

real **dirichlet_log**(vector *theta*, vector *alpha*)

The log of the Dirichlet density for simplex *theta* given prior counts (plus one) *alpha*

25.8. Vector Probabilities

The vector probability distributions have support on all of \mathbb{R}^K for some fixed K .

Multivariate Normal Distribution

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 $y \in \mathbb{R}^K$,

$$\text{MultiNormal}(y|\mu, \Sigma) = \frac{1}{(2\pi)^{K/2}} \frac{1}{\sqrt{|\Sigma|}} \exp\left(-\frac{1}{2}(y - \mu)^\top \Sigma^{-1} (y - \mu)\right),$$

where $|\Sigma|$ is the absolute determinant of Σ .

real **multi_normal_log**(vector *y*, vector *mu*, matrix *S*)

The log of the multivariate normal density of vector *y* given location *mu* and covariance matrix *S*

Multivariate Normal Distribution (Cholesky Parameterization)

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 $y \in \mathbb{R}^K$,

$$\text{MultiNormalCholesky}(y|\mu, L) = \text{MultiNormal}(y|\mu, LL^\top).$$

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*

Multivariate Student- t Distribution

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 $y \in \mathbb{R}^K$,

MultiStudentT($y|\nu, \mu, \Sigma$)

$$= \frac{1}{\pi^{K/2}} \frac{1}{\nu^{K/2}} \frac{\Gamma_x((\nu + K)/2)}{\Gamma(\nu/2)} \frac{1}{\sqrt{|\Sigma|}} \left(1 + \frac{1}{\nu} (y - \mu)^\top \Sigma^{-1} (y - \mu) \right)^{-(\nu+K)/2}.$$

real **multi_student_t_log**(vector y , real nu , vector mu , matrix S)

The log of the multivariate Student- t density of vector y given degrees of freedom nu , location mu , and scale matrix S

25.9. Covariance Matrix Distributions

The covariance matrix distributions have support on symmetric, positive-definite $K \times K$ matrices.

Wishart Distribution

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 $W \in \mathbb{R}^{K \times K}$,

$$\text{Wishart}(W|\nu, S) = \frac{1}{2^{\nu K/2}} \frac{1}{\Gamma_K(\frac{\nu}{2})} |S|^{-\nu/2} |W|^{(\nu-K-1)/2} \exp\left(\frac{1}{2} \text{tr}(S^{-1}W)\right),$$

where $\text{tr}()$ is the matrix trace function, and $\Gamma_K()$ is the multivariate Gamma function,

$$\Gamma_K(x) = \frac{1}{\pi^{K(K-1)/4}} \prod_{k=1}^K \Gamma\left(x + \frac{1-k}{2}\right).$$

real **wishart_log**(matrix W , real nu , matrix S)

The log of the Wishart density for positive-definite matrix W given degrees of freedom nu and scale matrix S

Inverse Wishart Distribution

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 $W \in \mathbb{R}^{K \times K}$,

$$\text{InvWishart}(W|\nu, S) = \frac{1}{2^{\nu K/2}} \frac{1}{\Gamma_K(\frac{\nu}{2})} |S|^{\nu/2} |W|^{-(\nu-K-1)/2} \exp\left(-\frac{1}{2} \text{tr}(SW^{-1})\right).$$

```
real inv.wishart_log(matrix W, real nu, matrix S)
```

The log of the inverse Wishart density for positive-definite matrix W given degrees of freedom nu and scale matrix S

LKJ Covariance Distribution

```
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.

25.10. 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.

LKJ Correlation Distribution

```
real lkj_corr_log(matrix y, real eta)
```

The log of the LKJ density for the correlation matrix y given nonnegative shape eta . The LKJ density is proportional to $\det(y)^{(\eta-1)}$. The shape parameter eta 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, especially as η becomes large. For $0 < \eta < 1$, the density has a trough at the identity matrix. See (?) for definitions.

LKJ Cholesky Distribution

```
real lkj_corr_cholesky_log(matrix L, real eta)
```

The log of the LKJ density for the lower-triangular Cholesky factor L of a correlation matrix given shape eta . The LKJ density is proportional to $\det(LL^\top)^{(\eta-1)} = \det(L)^{(2\eta-2)}$. See the previous subsection for details on interpreting the shape eta .

Part VI

Additional Topics

26. Bayesian Data Analysis

? 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.

26.1. Bayesian Modeling

(?) 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.

26.2. 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).

$$\begin{aligned} p(\theta|y) &= \frac{p(\theta, y)}{p(y)} && \text{[definition of conditional probability]} \\ &= \frac{p(y|\theta) p(\theta)}{p(y)} && \text{[chain rule]} \\ &= \frac{p(y|\theta) p(\theta)}{\int_{\Theta} p(y, \theta) d\theta} && \text{[law of total probability]} \\ &= \frac{p(y|\theta) p(\theta)}{\int_{\Theta} p(y|\theta) p(\theta) d\theta} && \text{[chain rule]} \\ &\propto p(y|\theta) p(\theta) && \text{[} y \text{ is fixed]} \end{aligned}$$

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

$$\Pr[\theta_k > 0] = \int_{\Theta} \mathbf{I}(\theta_k > 0) p(\theta|y) d\theta.$$

The indicator function, $\mathbf{I}(\phi)$, 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

$$\Pr[\tilde{y}_n > \tilde{y}_{n'}] = \int_{\Theta} \int_Y \mathbf{I}(\tilde{y}_n > \tilde{y}_{n'}) p(\tilde{y}|\theta) p(\theta|y) d\tilde{y} d\theta.$$

27. Markov Chain Monte Carlo Sampling

Like BUGS, Stan uses Markov chain Monte Carlo (MCMC) techniques to generate samples from the posterior distribution for inference.

27.1. 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 (?).

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 ellided and taken as given.

Now suppose it is possible to draw independent samples from $p(\theta)$ and let $\theta^{(1)}, \theta^{(2)}, \dots, \theta^{(N)}$ 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,

$$\lim_{N \rightarrow \infty} \hat{\mu} = \mu.$$

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.

27.2. 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 (?).

A Markov chain is a sequence of random variables $\theta^{(1)}, \theta^{(2)}, \dots$ where each variable is conditionally independent of all other variables given the value of the previous value. Thus if $\theta = \theta^{(1)}, \theta^{(2)}, \dots, \theta^{(N)}$, 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 27.5.

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.

27.3. 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 ? 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 ,

$$\bar{\theta}_m^{(\bullet)} = \frac{1}{N_m} \sum_{n=1}^{N_m} \theta_m^{(n)},$$

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 $\widehat{\text{var}}^+$ 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

$$\widehat{\text{var}}^+(\theta|y) = \frac{1}{M} \sum_{m=1}^M \frac{1}{N_m} \sum_{n=1}^{N_m} (\theta_m^{(n)} - \bar{\theta}_m^{(\bullet)})^2 + \frac{1}{M-1} \sum_{m=1}^M (\bar{\theta}_m^{(\bullet)} - \bar{\theta}_\bullet^{(\bullet)})^2.$$

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.

27.4. 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 $t \in \{0, 1, \dots\}$. The variograms are defined as follows for (univariate) samples $\theta_m^{(n)}$, where $m \in \{1, \dots, M\}$ 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.

27.5. 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 (?) for details of dual averaging and (?) 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 9 for examples.

Detailed Balance

HMC uses a Metropolis rejection step to ensure detailed balance of the resulting Markovian system; for details, see (?). NUTS uses a form of slice sampling which guarantees detailed balance; for details, see (?).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 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 27.4.

¹A 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

$$\pi(\omega') = \int_{\Omega} \pi(\omega) \phi(\omega'|\omega) d\omega.$$

28. 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 upper 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 support 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 include code to transform the parameters from unconstrained to constrained and apply the appropriate Jacobians.

28.1. 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 $f : \text{supp}(X) \rightarrow \mathbb{R}$ is a one-to-one, monotonic function 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 transformed 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

$$p_Y(y) = p_X(f^{-1}(y)) \left| \det J_{f^{-1}}(y) \right|,$$

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

$$J_{f^{-1}}(y) = \begin{bmatrix} \frac{\partial x_1}{\partial y_1} & \cdots & \frac{\partial x_1}{\partial y_K} \\ \vdots & \vdots & \vdots \\ \frac{\partial x_K}{\partial y_1} & \cdots & \frac{\partial x_K}{\partial y_K} \end{bmatrix}.$$

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.

28.2. 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

$$\left| \frac{d}{dy} (\exp(y) + a) \right| = \exp(y).$$

Therefore, given the density p_X of X , the density of Y is

$$p_Y(y) = p_X(\exp(y) + a) \cdot \exp(y).$$

28.3. 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

$$\left| \frac{d}{dy} (b - \exp(y)) \right| = \exp(y).$$

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

$$p_Y(y) = p_X(b - \exp(y)) \cdot \exp(y).$$

28.4. 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

$$\frac{d}{dy} \text{logit}^{-1}(y) = \text{logit}^{-1}(y) \cdot (1 - \text{logit}^{-1}(y)).$$

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

$$\left| \frac{d}{dy} (a + (b - a) \cdot \text{logit}^{-1}(y)) \right| = (b - a) \cdot \text{logit}^{-1}(y) \cdot (1 - \text{logit}^{-1}(y)).$$

Therefore, the density of the transformed variable Y is

$$p_Y(y) = p_X(a + (b - a) \cdot \text{logit}^{-1}(y)) \cdot (b - a) \cdot \text{logit}^{-1}(y) \cdot (1 - \text{logit}^{-1}(y)).$$

Despite the apparent complexity of this expression, most of the terms are repeated and thus only need to be evaluated once. Most importantly, $\text{logit}^{-1}(y)$ only needs to be evaluated once, so there is only one call to $\exp(-y)$.

28.5. 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 10.6).

In constraint terms, an ordered K -vector $x \in \mathbb{R}^K$ satisfies

$$x_k < x_{k+1}$$

for $k \in \{1, \dots, K-1\}$.

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

$$p_Y(y) = p_X(f^{-1}(y)) \prod_{k=2}^K \exp(y_k).$$

28.6. 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 $1 \leq k \leq K$,

$$x_k > 0,$$

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, \dots, 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

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

¹For an alternative derivation of the same transform using hyperspherical coordinates, see (?).

(x_1, \dots, x_{K-1}) . Label what's left of the original stick x_K . The vector $x = (x_1, \dots, x_K)$ 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 $1 \leq k < K$ by

$$z_k = \text{logit}^{-1} \left(y_k - \log \left(\frac{1}{K - k} \right) \right).$$

The logit term in the above definition adjusts the transform so that a zero vector y is mapped to the simplex $x = (1/K, \dots, 1/K)$. 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 $1 \leq k < K$ 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, \dots, 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,

$$z_k = \frac{x_k}{1 - \sum_{k'=1}^{k-1} x_{k'}}.$$

28.7. Correlation Matrices

A $K \times K$ correlation matrix x must be is a symmetric, so that

$$x_{k,k'} = x_{k',k}$$

for all $k, k' \in \{1, \dots, K\}$, it must have a unit diagonal, so that

$$x_{k,k} = 1$$

for all $k \in \{1, \dots, K\}$, 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 ?. 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 $\tanh : \mathbb{R} \rightarrow (0, 1)$

$$\tanh x = \frac{\exp(2x) - 1}{\exp(2x) + 1}.$$

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

$$w_{i,j} = \begin{cases} 0 & \text{if } i > j, \\ 1 & \text{if } 1 = i = j, \\ \prod_{i'=1}^{i-1} (1 - z_{i',j}^2)^{1/2} & \text{if } 1 < i = j, \\ z_{i,j} & \text{if } 1 = i < j, \text{ and} \\ z_{i,j} \prod_{i'=1}^{i-1} (1 - z_{i',j}^2)^{1/2} & \text{if } 1 < i < j. \end{cases}$$

This does not require as much computation per matrix entry as it may appear; calculating the rows in terms of earlier rows yields the more manageable expression

$$w_{i,j} = \begin{cases} 0 & \text{if } i > j, \\ 1 & \text{if } 1 = i = j, \\ z_{i,j} & \text{if } 1 = i < j, \text{ and} \\ z_{i,j} w_{i-1,j} (1 - z_{i-1,j}^2)^{1/2} & \text{if } 1 < i \leq j. \end{cases}$$

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

$$\det x = \prod_{i=1}^{K-1} \prod_{j=i+1}^K (1 - z_{i,j}^2) = \prod_{1 \leq i < j \leq K} (1 - z_{i,j}^2),$$

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

$$z_{i,j} = \begin{cases} 0 & \text{if } i \leq j, \\ w_{i,j} & \text{if } 1 = i < j, \text{ and} \\ w_{i,j} \prod_{i'=1}^{i-1} (1 - z_{i',j}^2)^{-2} & \text{if } 1 < i < j. \end{cases}$$

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.

28.8. 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 $z = \text{chol}(x)$ 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 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

$$x = z z^\top.$$

²An 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

$$\Sigma = \text{diag}(\sigma) \times \Omega \times \text{diag}(\sigma),$$

so that each entry is just a deviation-scaled correlation,

$$\Sigma_{m,n} = \sigma_m \times \sigma_n \times \Omega_{m,n}.$$

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

$$p_Y(y) = p_X(f^{-1}(y)) 2^K \prod_{k=1}^K z_{k,k}^{K-k+2}.$$

Appendices

A. Licensing

Stan and its two dependent libraries, Boost and Eigen, are distributed under liberal freedom-respecting¹ 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>

¹The link <http://www.gnu.org/philosophy/open-source-misses-the-point.html> leads to a discussion about terms “open source” and “freedom respecting.”

²See <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).

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-src-1.m.p.tgz` (`m` is the minor version and `p` the patch level) from the Stan downloads list,

<http://code.google.com/p/stan/downloads/list>

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 `.tgz` file into file `stan-src-1.m.p.tar`, double-click the `.tgz` file to unpack.

Double click on the `.tar` file to unarchive directory `stan-src-1.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-src-1.m.p.tgz`, where `m` is the minor version and `p` the patch level), from the Stan downloads page,

```
http://code.google.com/p/stan/downloads/list
```

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-src-1.m.p
```

with

```
> tar -xzf stan-src-1.m.p.tgz
```

B.4. Step-by-Step Windows Install Instructions

Stan has been tested on Windows XP and Windows 7.

Note for Cygwin Users

Stan can be run under Cygwin, the Linux 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 Linux.

The remaining instructions in this section are for users wishing to run Stan under Windows in a command (i.e., DOS) shell.

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.

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-src-1.m.p.tgz`, where `m` is the minor version and `p` the patch level.

Download the latest Stan source from the Stan downloads page,

<http://code.google.com/p/stan/downloads/list>

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 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 `.tgz` 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-src-1.m.p.tgz
```

The `--no-same-owner` flag is not strictly necessary, but it removes a bunch of irrelevant warnings.

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: 1.51.0

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: 3.1.1

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, 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

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.

Mac OS X “Snow Leopard” or earlier Xcode 3: Good luck; we couldn't find it

Mac OS X “Lion” or later Xcode 4: <https://developer.apple.com/xcode/>

Then, 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.

More Recent Compilers

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.”

L^AT_EX Typesetting Package

Stan uses the L^AT_EX 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 BUGS¹ 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 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).
- 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

¹Except where otherwise noted, we use "BUGS" to refer to WinBUGS, OpenBUGS, and JAGS, indiscriminately.

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 counterindicate 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:

<i>BUGS</i>	<i>Stan</i>
<code>dnorm</code>	<code>normal</code>
<code>dbinom</code>	<code>binomial</code>
<code>dpois</code>	<code>poisson</code>
<code>⋮</code>	<code>⋮</code>

- 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:

```
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.

²Most distributions have been vectorized, but currently the truncated versions may not exist and may not be vectorized.

- Stan supports general conditional statements using a standard if-else syntax. For example, a zero-inflated (or -deflated) Poisson mixture model may be defined using an array of mixture component indicators z as follows.

```

data {
  int<lower=0> N;
  int<lower=0> y[N];
  ...
}
model {
  for (n in 1:N) {
    if (y[n] == 0)
      y[n] ~ bernoulli(theta);
    } else {
      y[n] ~ poisson(lambda) T[1,];
    }
  }
}

```

- 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 (?). 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, $\text{Norm}(y|0, 1) \times \text{Norm}(y|2, 3)$, 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, $\text{Norm}(x|0.8y, \sigma) \times \text{Norm}(y|0.8x, \sigma)$, 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 (?) and (?) 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.

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.

¹Even 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.

```

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 16.1.

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-spaces of indentation, like the usual standard used for C/C++ (and unlike the usual standard for Java, which is four spaces, or Python, which is one or more tabs depending on indent level). The other sensible choice is four characters. Just be consistent.

Space Between `if` and Condition

Use a space after `ifs`. 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).

```
lp__ <- lp__ + (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,-2));
```

²This 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.

E. Auxiliary Stan Tools

The Stan development team mainly uses command-line programs and standalone text editors like `vi` and `emacs`.

E.1. Tools for Emacs

Stan's emacs mode is defined relative to the directory `<stan-home>` in which Stan was unpacked from source.

```
<stan-home>/src/StanMode/stan-mode.el
```

Installing Emacs Mode

Typically, emacs is set up to read a `.emacs` file from the user's home directory. Add the following two lines to this file replacing `<stan-home>` with the path to where Stan was unpacked.

```
(add-to-list 'load-path "/Users/carp/stan/src/StanMode/")
(require 'stan-mode)
```

For Aquamacs on Mac OS X, these two lines can be placed in the `.emacs` file or in the following preferences file.

```
~/Library/Preferences/AquamacsEmacs/Preferences.el
```

Index

- Phi**
(real *x*):real, 139
- abs**
(int *x*):int, 131
(real *x*):real, 136
- acos**
(real *x*):real, 138
- acosh**
(real *x*):real, 138
- asin**
(real *x*):real, 138
- asinh**
(real *x*):real, 138
- atan**
(real *x*):real, 138
- atan2**
(real *x*, real *y*):real, 138
- atanh**
(real *x*):real, 138
- bernoulli.log**
(ints *y*, reals *theta*):real, 153
- bernoulli.logit.log**
(ints *y*, reals *alpha*):real, 153
- beta.binomial.log**
(ints *n*, ints *N*, reals *alpha*,
reals *beta*):real, 154
- beta.log**
(reals *theta*, reals *alpha*, reals
beta):real, 161
- binary.log.loss**
(int *y*, real *y.hat*):real, 139
- binomial.coefficient.log**
(real *x*, real *y*):real, 140
- binomial.log**
(int *n*, int *N*, real *theta*):real,
154
- block**
(matrix *x*, int *i*, int *j*, int
n.rows, int *n.cols*):matrix,
150
- categorical.log**
(int *y*, vector *theta*):real, 154
- cauchy.log**
(reals *y*, reals *mu*, reals
sigma):real, 158
- cbrt**
(real *x*):real, 137
- ceil**
(real *x*):real, 136
- chi.square.log**
(reals *y*, reals *nu*):real, 159
- cholesky.decompose**
(matrix *A*):matrix, 152
- col**
(matrix *x*, int *n*):vector, 149
- cols**
(matrix *x*):int, 142
(row.vector *x*):int, 142
(vector *x*):int, 142
- cos**
(real *x*):real, 138
- cosh**
(real *x*):real, 138
- crossprod**
(matrix *x*):matrix, 147
- cumulative.sum**
(real[] *x*):real[], 146
(row.vector *rv*):row.vector, 146
(vector *v*):vector, 146
- determinant**
(matrix *A*):real, 151
- diag.matrix**

(vector x):matrix, 149	(matrix x):matrix, 146
diag_post_multiply	(real x):real, 137
(matrix m, row_vector	(row_vector x):row_vector, 146
rv):matrix, 147	(vector x):vector, 146
(matrix m, vector v):matrix, 147	exp2
	(real x):real, 137
diag_pre_multiply	expm1
(row_vector rv, matrix	(real x):real, 140
m):matrix, 147	exponential.cdf
(vector v, matrix m):matrix, 147	(real y, real beta):real, 160
	exponential.log
diagonal	(reals y, reals beta):real, 160
(matrix x):vector, 149	fabs
dirichlet.log	(real x):real, 136
(vector theta, vector	fdim
alpha):real, 162	(real x, real y):real, 136
dot_product	floor
(row_vector x, row_vector	(real x):real, 136
y):real, 146	fma
(row_vector x, vector y):real,	(real x, real y, real z):real,
146	140
(vector x, row_vector y):real,	fmax
146	(real x, real y):real, 136
(vector x, vector y):real, 146	fmin
dot_self	(real x, real y):real, 136
(row_vector x):real, 147	fmod
(vector x):real, 146	(real x, real y):real, 136
double_exponential.log	gamma.log
(reals y, reals mu, reals	(reals y, reals alpha, reals
sigma):real, 158	beta):real, 160
e	hypergeometric.log
() :real, 132	(int n, int N, int a, int
eigenvalues.sym	b):real, 154
(matrix A):vector, 151	hypot
eigenvectors.sym	(real x, real y):real, 137
(matrix A):matrix, 151	if_else
epsilon	(int cond, real x, real y):real,
() :real, 132	135
erf	int_step
(real x):real, 139	(int x):int, 131
erfc	inv_chi_square.log
(real x):real, 139	
exp	

```

    (reals y, reals nu):real, 159

inv_cloglog
    (real x):real, 139

inv_gamma_log
    (reals y, reals alpha, reals
     beta):real, 160

inv_logit
    (real x):real, 139

inv_wishart_log
    (matrix W, real nu, matrix
     S):real, 164

inverse
    (matrix A):matrix, 151

lbeta
    (real x, real y):real, 140

lgamma
    (real x):real, 139

lkj_corr_cholesky_log
    (matrix L, real eta):real, 164

lkj_corr_log
    (matrix y, real eta):real, 164

lkj_cov_log
    (matrix W, vector mu, vector
     sigma, real eta):real, 164

lmgamma
    (int n, real x):real, 139

log
    (matrix x):matrix, 145
    (real x):real, 137
    (row_vector x):row_vector, 145
    (vector x):vector, 145

log10
    ():real, 132
    (real x):real, 137

log1m
    (real x):real, 140

log1m_inv_logit
    (real x):real, 140

log1p
    (real x):real, 140

log1p_exp
    (real x):real, 140

log2
    ():real, 132
    (real x):real, 137

log_inv_logit
    (real x):real, 140

log_sum_exp
    (real x, real y):real, 140
    (real x[]):real, 141

logistic_log
    (reals y, reals mu, reals
     sigma):real, 158

logit
    (real x):real, 139

lognormal_cdf
    (real y, real mu, real
     sigma):real, 159

lognormal_log
    (reals y, reals mu, reals
     sigma):real, 159

max
    (int x, int y):int, 131
    (int x[]):int, 141
    (matrix x):real, 148
    (real x[]):real, 141
    (row_vector x):real, 148
    (vector x):real, 148

mdivide_left_tri_low
    (matrix a, matrix b):matrix, 151
    (matrix a, vector b):vector, 151

mdivide_right_tri_low
    (matrix b, matrix a):matrix, 151
    (row_vector b, matrix
     a):row_vector, 151

mean
    (matrix x):real, 149
    (real x[]):real, 141
    (row_vector x):real, 149
    (vector x):real, 148

min
    (int x, int y):int, 131
    (int x[]):int, 141
    (matrix x):real, 148

```

```

(real x[]):real, 141
(row_vector x):real, 148
(vector x):real, 148

multi_normal_cholesky_log
(vector y, vector mu, matrix
  L):real, 162

multi_normal_log
(vector y, vector mu, matrix
  S):real, 162

multi_student_t_log
(vector y, real nu, vector mu,
  matrix S):real, 163

multinomial_log
(int[] y, vector theta):real,
  156

multiply_log
(real x, real y):real, 140

multiply_lower_tri_self_transpose
(matrix x):matrix, 147

neg_binomial_log
(ints n, reals alpha, reals
  beta):real, 155

negative_epsilon
():real, 132

negative_infinity
():real, 132

normal_cdf
(reals y, reals mu, reals
  sigma):real, 157

normal_log
(reals y, reals mu, reals
  sigma):real, 157

not_a_number
():real, 132

operator
=
(int x, int y):int, 133
(real x, real y):int, 134

operator'
(matrix x):matrix, 150
(row_vector x):vector, 150

```

```

(vector x):row_vector, 150

operator*
(int x, int y):int, 131
(matrix x, matrix y):matrix, 144
(matrix x, real y):matrix, 144
(matrix x, vector y):vector, 144
(real x, matrix y):matrix, 143
(real x, real y):real, 135
(real x, row_vector
  y):row_vector, 143
(real x, vector y):vector, 143
(row_vector x, matrix
  y):row_vector, 144
(row_vector x, real
  y):row_vector, 143
(row_vector x, vector y):real,
  143
(vector x, real y):vector, 143
(vector x, row_vector y):matrix,
  143

```

```

operator+
(int x):int, 131
(int x, int y):int, 131
(matrix x, matrix y):matrix, 143
(matrix x, real y):matrix, 144
(real x):real, 136
(real x, matrix y):matrix, 144
(real x, real y):real, 135
(real x, row_vector
  y):row_vector, 144
(real x, vector y):vector, 144
(row_vector x, real
  y):row_vector, 144
(row_vector x, row_vector
  y):row_vector, 143
(vector x, real y):vector, 144
(vector x, vector y):vector, 143

```

```

operator-
(int x):int, 131
(int x, int y):int, 131
(matrix x):matrix, 142
(matrix x, matrix y):matrix, 143
(matrix x, real y):matrix, 145
(real x):real, 135
(real x, matrix y):matrix, 145
(real x, real y):real, 135
(real x, row_vector
  y):row_vector, 145
(real x, vector y):vector, 144

```

```

(row_vector x):row_vector, 142
(row_vector x, real
    y):row_vector, 144
(row_vector x, row_vector
    y):row_vector, 143
(vector x):vector, 142
(vector x, real y):vector, 144
(vector x, vector y):vector, 143

operator.*
(matrix x, matrix y):matrix, 145
(row_vector x, row_vector
    y):row_vector, 145
(vector x, vector y):vector, 145

operator./
(matrix x, matrix y):matrix, 145
(row_vector x, row_vector
    y):row_vector, 145
(vector x, vector y):vector, 145

operator/
(int x, int y):int, 131
(real x, real y):real, 135
(row_vector b, matrix
    A):row_vector, 150

operator<
(int x, int y):int, 133
(real x, real y):int, 133

operator<=
(int x, int y):int, 133
(real x, real y):int, 133

operator==
(int x, int y):int, 133
(real x, real y):int, 134

operator>
(int x, int y):int, 133
(real x, real y):int, 133

operator>=
(int x, int y):int, 133
(real x, real y):int, 134

operator\
(matrix A, vector b):vector, 150

operator
    (int x):int, 134
    (real x):int, 134

operator&&
    (int x, int y):int, 134
    (real x, real y):int, 134

ordered.logistic.log
    (int k, real eta, vector
        c):real, 155

pareto.log
    (reals y, reals y_min, reals
        alpha):real, 161

pi
    ():real, 132

poisson.log
    (ints n, reals lambda):real, 155

poisson.log.log
    (ints n, reals alpha):real, 156

positive.infinity
    ():real, 132

pow
    (real x, real y):real, 137

prod
    (int x[]):real, 141
    (matrix x):real, 148
    (real x[]):real, 141
    (row_vector x):real, 148
    (vector x):real, 148

round
    (real x):real, 137

row
    (matrix x, int m):row_vector,
        149

rows
    (matrix x):int, 142
    (row_vector x):int, 142
    (vector x):int, 142

scaled.inv.chi.square.log
    (reals y, reals nu, reals
        s):real, 159

sd
    (matrix x):real, 149

```



```

    (real x[]):real, 141
    (row_vector x):real, 149
    (vector x):real, 149

sin
    (real x):real, 138

singular_values
    (matrix A):vector, 152

sinh
    (real x):real, 138

softmax
    (vector x):vector, 150

sqrt
    (real x):real, 137

sqrt2
    ():real, 132

square
    (real x):real, 137

step
    (real x):real, 135

student_t_log
    (reals y, reals nu, reals mu,
     reals sigma):real, 158

sum
    (int x[]):real, 141
    (matrix x):real, 148
    (real x[]):real, 141
    (row_vector x):real, 148
    (vector x):real, 148

tan
    (real x):real, 138

tanh
    (real x):real, 138

tcrossprod
    (matrix x):matrix, 147

tgamma
    (real x):real, 139

trace
    (matrix A):real, 151

trunc
    (real x):real, 137

uniform_log
    (reals y, reals alpha, reals
     beta):real, 161

variance
    (matrix x):real, 149
    (real x[]):real, 141
    (row_vector x):real, 149
    (vector x):real, 149

weibull_cdf
    (real y, real alpha, real
     sigma):real, 160

weibull_log
    (reals y, reals alpha, reals
     sigma):real, 160

wishart_log
    (matrix W, real nu, matrix
     S):real, 163

```