Stan Modeling Language

Stan's Development Team

Version 1.0

April 1, 2012

Abstract

Stan is a general-purpose probabilistic modeling framework designed to support full Bayesian inference. This document describes Stan's modeling language for specifying the joint probability of observed data and unobserved parameters. Stan's compiler parses model specifications and generates C++ code. Stan's modeling language is based on BUGS and JAGS, and like them, allows users to develop, fit, and evaluate Bayesian models without knowing C++.

Stan performs inference by Markov chain Monte Carlo (MCMC) sampling of parameter values from the posterior distribution of parameters given the observed data. Stan employs the no-U-turn sampler (NUTS), an adaptive form of Hamiltonian Monte Carlo (HMC) that alleviates the need for user tuning of HMC's rather sensitive parameters. By making effective use of the gradient of the log posterior, HMC converges and explores the parameter distribution faster than Gibbs sampling or random-walk Metropolis-Hastings. The improved sampling algorithm and more compact and efficient compiled code allows Stan to operate on data of larger scales and models of more complex structure than BUGS or JAGS.

Contents

Part I

Introduction

1. What is Stan?

This document is a reference manual and getting started guide for using Stan's probabilistic modeling language. After describing the overall system in this introduction and providing a hands-on quick-start guide in the following chapter, the remainder of the document is devoted to fully documenting the behavior of Stan's modeling language.

Stan's modeling language and its execution behavior are similar to that of its progenitors, BUGS and JAGS. It differs in many particulars of the modeling language, which is more like an impertative programming language than the declarative specifications of BUGS and JAGS.

1.1. Stan's Modeling Language

Stan's modeling language allows users to code a Bayesian model specifying a joint probability function

$$p(y, \theta),$$

where

- y is a vector of known values, such as constants, hyperparameters, and modeled data, and
- θ is a vector of unknown values, such as estimated parameters, missing data, and simulated values.

To simplify terminology, y will be called the data vector and θ the parameter vector. The probability function $p(y, \theta)$ need only be specified up to a multiplicative constant with respect to any fixed data vector y. This ensures proportionality of the posterior to the specified joint probability,

$$p(\theta|y) \propto p(y, \theta) = p(\theta|y)\, p(y).$$

Stan's language is more imperative than its declarative predecessors, BUGS and JAGS. Statements are executed in the order they are specified and variables and expressions are strongly typed and declared as data or parameters in the model rather than by a calling function. Stan also supports a broader range of arithmetic, matrix, and linear algebra operations than BUGS or JAGS. Users may manipulate log probability functions directly and are not required to use proper priors.

1.2. Stan's Compiler

Stan's compiler, `stanc`, reads a user program in Stan's modeling language and generates a C++ class implementing the model specified by that program. Stan automatically applies a multivariate transform (and its Jacobian determinant) to free any constrained parameters, such as deviations (constrained to be positive), simplexes (a vector constrained to be positive and to sum to 1), and covariance matrices (positive definiteness). The result is an unconstrained sampling (or optimization) problem from the perspective of the sampler. From the user's perspective, this transform happens behind the scenes, driven by the types declared for each of the parameters.

The generated C++ class can be plugged into Stan's continuous and discrete samplers to read the data y and then draw a sequence of sample parameter vectors $\theta^{(m)}$ according to the posterior,

$$p(\theta|y) = \frac{p(y,\theta)}{p(y)} \propto p(y,\theta).$$

The resulting samples may be used for full Bayesian inference, much of which can be carried out within Stan.

1.3. Stan's 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 and uses a Metropolis adjustment to ensure detailed balance of the resulting Markovian system. 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.

HMC treats the position of a particle, log probability as a negative potential energy function, then samples by adding random kinetic energy and simulating the

In addition to basic HMC, Stan implements an adaptive version of HMC, the No-U-Turn Sampler (NUTS). NUTS automatically tunes step sizes and a diagonal mass matrix during warmup and then adapts the number of leapfrog integration steps during sampling. Stan is expressive enough to allow most discrete variables to be marginalized out. For the remaining discrete parameters, Stan uses Gibbs sampling if there are only a few outcomes and adaptive slice sampling otherwise.

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. For installation information, see Appendix A. Later chapters are devoted to expanding on the material in this chapter with full reference documentation.

2.1. A Minimal Program

Stan is distributed with several working models. The simplest of these is found in the following location relative to the top-level distribution.

```
src/models/basic_distributions/normal.stan
```

The contents of this file are as follows.

```
parameters {
  real y;
}
model {
  y ~ normal(0,1);
}
```

The model's single parameter `y` is declared to take real values. The probability model specifies that `y` has a normal distribution with location 0 and scale 1. Basically, this model will sample a single unit normal variate.

2.2. Whitespace and Semicolons

In Stan, every variable declaration and atomic statement must be terminated by a semicolon (`;`). This is the convention followed by programming languages such as C++. It is not the convention followed by the statistical languages R, BUGS, or JAGS.

The reason for the C++ convention is to ensure that differences in whitespace are not meaningful. In R, BUGS, and JAGS, the following is a complete, legal statement.

```
a <- b +
     c
```

In contrast, the usual way of typesetting mathematics and laying out code in programming languages, with the operator continuing the expression beginning a new line, is invalid.

```
a <- b
     + c
```

The only difference is in the kind of whitespace between `b` and `+` and between `+` and `c`. In Stan, there is no whitespace-dependent behavior. Neither of these is a complete statement, whereas either one terminated with a semicolon is. The second form is recommended for C++ and Stan.

2.3. Compiling with `stanc`

Starting at Stan's home directory, written here as `$stan`, the model may be compiled by the Stan compiler, `stanc`, into C++ code as follows.

```
> cd $stan
> stanc src/models/basic_distributions/normal.stan

Model name=anon_model
Input file=src/models/basic_distributions/normal.stan
Output file=anon_model.cpp
```

The output indicates the name of the model, here the default value `anon_model`, the input file from which the Stan program is read, here `normal.stan`, and the output file to which the generated C++ code is written, here `anon_model.cpp`. See Chapter 3 for more documentation on the `stanc` compiler.

2.4. Compiling the Generated Code

The file generated by `stanc` must next be compiled with a C++ compiler by linking to Stan's source and library directories using the `-I` option of the compiler. The following example uses the `clang++` compiler for C++.

```
> clang++ -I src -I lib anon_model.cpp
```

This command invokes the `clang++` compiler for C++ to create a platform-specific executable in the default location, which is `a.out` by convention. If all goes well, as above, there is no output to the console. More information about compiling the C++ code generated by Stan may be found in Chapter 4. Installation information for C++ compilers may be found in Appendix A.

2.5. Running the Sampler

The executable resulting from compiling the generated C++ may be run as follows.

```
> ./a.out

STAN SAMPLING COMMAND
data =
init = random initialization
samples = samples.csv
append_samples = 0
seed = 1331941513 (randomly generated)
chain_id=1 (default)
iter = 2000
warmup = 1000
thin = 1
leapfrog_steps = -1
max_treedepth = 10
epsilon = -1
epsilon_pm = 0
epsilon_adapt_off = 0
delta = 0.5
gamma = 0.05

Iteration: 2000 / 2000 [100%]  (Sampling)
```

The program indicates to the standard output that the samples are written to `samples.csv`. The first few lines of this file are comments about aspects of the run.

```
> cat samples.csv

# Samples Generated by Stan
#
# stan_version_major=alpha
# stan_version_minor=0
# data=
# init=random initialization
# append_samples=0
# seed=1331941796
# chain_id=1
# iter=2000
# warmup=1000
# thin=1
# leapfrog_steps=-1
# max_treedepth=10
# epsilon=-1
# epsilon_pm=0
# delta=0.5
# gamma=0.05
...
```

The ellipses notation, `...`, indicates that the output continues beyond what's shown. Here, what follows is the data in standard comma-separate value (CSV) notation.

```
...
lp__,treedepth__,y
-0.0126699,1,0.159185
-0.222796,1,-0.667527
-0.222796,1,-0.667527
-0.404457,1,-0.899397
...
```

The first line consists of a header indicating the names of the variables on the lines to follow, and each following line indicates a single sampled value of the parameters. The first column is reserved for the (unnormalized) log probability (density) of the parameters, with name `lp__` (the underscores are to prevent name conflicts with user-defined model parameters). The next values are for reporting the behavior of the sampler. In this case, the NUTS sampler was used, so there is a report of the depth of tree it explored, with variable name `treedepth__`. The remaining values are parameters. Here, the model has only one parameter, `y`. The first sampled value for `y` is 0.159185, the second is -0.667527, and so on.

Note that the second sampled value is repeated. This is not a bug. Rather, it is the behavior to expect from a sampler using a Metropolis acceptance step for proposals, as Stan's samplers HMC and NUTS do.

2.6. Data

Stan allows data to be specified in programs, used in models, and read into compiled Stan programs. This section provides an example of coding and running a Stan program with data stored in a file in the S/R dump format.

The Stan program in

```
src/models/basic_estimators/bernoulli.stan
```

can be used to estimate a Bernoulli parameter `theta` from `N` binary observations. The file contains the following code.

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

This program declares two data variables in its `data` block. The first data variable, `N`, is an integer encoding the number of observations. The declaration `int(0,)` indicates that `N` must take on non-negative values. The second data variable, `y`, is declared as `y[N]`, specifying that it is an array of `N` values. Each of these values has the declared type, `int(0,1)`, an integer between 0 and 1 inclusive, i.e., a binary value. The `N` individual binary values in the array `y` are accessed using standard array notation, indexing from 1, as `y[1]`, `y[2]`, ..., `y[N]`.

The `parametes` block declares a single parameter, `theta`. Its type is given as `real(0,1)`, meaning it takes on continuous values between 0 and 1 inclusive. The constraint is necessary in order to ensure that `theta` takes on a legal value as the success parameter in the Bernoulli distribution in which it is used in the `model` block of the program.

The `model` block consists of a for-loop for the data. The loop is specified so that the body is executed for values of `n` between `1` and `N` inclusive. The body here is a sampling statement specifying that the variable `y[n]` is modeled as having a Bernoulli distribution with parameter `theta`.

A sample data file for this program can be found in the file `bernoulli.Rdata` in the same directory. This data file has the following contents.

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

A data file must contain appropriate values for all of the data variables declared in the Stan program's `data` block. Here there is a non-negative integer value for `N` and an array of length `N` (i.e., 10) integer values between 0 and 1 inclusive. The array is coded using the `S` sequence notation `c(...)`. The dump format supported by Stan is documented in Chapter 6.

The program is compiled by `stanc` and the C++ compiler in the same way. This time, the output model gets an explicitly specified name.

```
> stanc --name=bern src/models/basic_estimators/bernoulli.stan

Model name=bern
Input file=src/models/basic_estimators/bernoulli.stan
Output file=bern.cpp
```

As before, the C++ compiler needs to be given the name of generated file.

```
> clang++ -O3 -I src -I lib -o bern bern.cpp
```

There are two new compiler options here. The option `-O3` sets optimization to level 3, which generates much faster executable code at the expense of slower compilation. The name of the executable is also specified, using the option `-o bern`. Now the code may be executed by calling its executable with the data file specified.

```
> ./bern --data=src/models/basic_estimators/bernoulli.Rdata
```

2.7. Proper and Improper Priors

The model in the previous section does not contain a sampling statement for `theta`. The default behavior is to give `theta` a uniform prior. In this case, a uniform prior is proper because `theta` is bounded to a finite interval. Improper priors are also allowed in Stan programs; they arise from unconstrained parameters without sampling statements. The uniform prior could have also been added explicitly by adding the following statement to the `model` block of the program.

```
theta ~ uniform(0,1);
```

A third way to specify that `theta` has a uniform distribution between 0 and 1 is with the beta distribution.

```
theta ~ beta(1,1);
```

The beta distribution is conjugate to the Bernoulli, but Stan (at least as of yet) does not make use of this information. On the other hand, these three approaches, no prior, uniform prior, and beta prior, are equally efficient in Stan's sampler, because their uniformity can be determined at compile time and thus computations related to them eliminated. There is further discussion of Stan optimization in Chapter 19

Part II

Commands and Data Formats

3. Compiling Stan Programs to C++

Preparing a Stan program to be run involves two compilation steps,

1. compiling the Stan program to C++, and
2. compiling the resulting C++ to an executable.

This chapter discusses the first step; the second step is discussed in Chapter 4.

3.1. The `stanc` Compiler

The `stanc` compiler converts Stan programs to C++ programs.

The first thing it does is parse the Stan program. If the parser is successful, it then generates C++ code. If the parser fails, it will provide an error message indicating where and why the error occurred.

The following example illustrates a fully qualified call to `stanc`.

```
> stanc --name=binary_normal --o=binorm.cpp binormal.stan
```

This call specifies the name of the model, here `binary_normal`. This will determine the name of the class implementing the model in the C++ code. The C++ code implementing the class is written to `binorm.cpp`. The final argument, `binormal.stan`, is the file from which to read the Stan program.

3.2. Command-Line Options

`--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 = anon_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`

4. Compiling C++ Programs

Stan has been developed using two portable, open-source compilers, `g++` and `clang++`, which run under Windows, Macintosh, and Unix/Linux. Stan has also been compiled using MSVC, a Windows-specific compiler from Microsoft.

4.1. Which Compiler?

It has been our experience that `clang++` is much faster to compile at all optimization levels than `g++`, but that the code generated by `g++` is slightly faster to execute.

4.2. What the Compiler Does

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

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

4.3. Including Library Code

Stan is written as a set of header-only libraries. This simplifies writing code that uses Stan. The only thing that needs to be done is to include the relevant libraries.

The compiler command-line option to include a header-only library is

```
-I path-to-library.
```

The path to the library must be such that any `#include` statements within the C++ source files be resolvable starting from the path to the library.

The header-only library code for Stan itself is located in the subdirectory `src/` of the top-level Stan directory. To allow programs to use the Stan library, the compiler needs to be given the option

```
-I stan/src
```

where *stan* is the path to the top-level Stan directory. If the compiler is called from the top-level Stan directory, it suffices to use `-I src`, as in the examples in the first chapter.

Stan depends on two open-source libraries,

1. Boost general purpose C++ libraries, and
2. Eigen matrix and linear algebra C++ libraries

These are both distributed along with Stan in the directory *stan*`/lib/`, where again *stan* is the top-level directory of the Stan distribution. Both libraries take include paths starting under `lib/`. For most uses of Stan, it is also necessary to include an explicit compiler option to include Eigen and Boost,

```
-I stan/lib
```

with *stan* being the location of the top-level Stan directory. Thus calling Stan typically requires all three of Stan, Boost, and Eigen to be included, which is accomplished with

```
-I stan/lib -I stan/src
```

where *stan* is the path to the top-level Stan directory.

4.4. Compiler Optimization

Stan was written with an optimizing compiler in mind. For that reason, it runs as much as an order of magnitude or more faster with optimization turned on.

For development, we recommend optimization level 0, whereas for sampling, we recommend optimization level 3. These are controlled through the compiler option `-O` (capital letter 'O'). To generate efficient code, use

```
-O3
```

where the first character is the capital letter 'O'. For faster compile time but less efficient code, use

```
-O0
```

where the first character is the capital letter 'O' and the second character is the digit '0'.

4.5. Executable Name

If no name is provided for the executable, the default value of `a.out` is used. This executable will show up in the directory from which the compiler was called.

To put the executable in a different location, the `-o` *path-to-executable* command may be used. In an earlier example, `-o bern` was used to write the executable to a file called `bern`. (In Windows, executables are suffixed with `.exe`.)

5. Running a Stan Program

Once a Stan program has been compiled (see Chapter 3),

6. Dump Data Format

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

6.1. Scalar Variables

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

```
y <-
17.2
```

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

6.2. 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)
```

It is possible to define an array without a declaration of dimensionality because the reader just counts the number of entries to determine the size of the array.

6.3. 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.

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

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

6.4. Integer- and Real-Valued Variables

There is no declaration in a dump file that distinguishes integer versus real values. If a value in a dump file's definition of a variable contains a decimal point, Stan assumes that the values are real. If there are no decimal points in a variable's defined value, the value may be assigned to variables declared as integer or real in Stan.

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.

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` delcared as real in Stan.

6.5. Quoted Variable Names

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

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

6.6. 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
```

Following its roots in R, breaking before the assignment arrow is not allowed.

```
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 $1 \times 2 \times 3$ array is valid.

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

6.7. General R Sequence Syntax

Sometimes, R will use shorthand for its output. For example, starting R,

```
> R
```

and then within R, executing the following commands,

```
R> e <- matrix(c(1,2,3,4,5,6),nrow=2,ncol=3)
R> dump("e")
```

leads to a `dumpdata.R` file being created with the following contents.

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

R has used the fact that it allows a contiguous sequence to be specified with its start and end point using the notation `2:3`. Stan cannot currently parse this format of input.

Alternatively, R is prone to include long-integer specifiers. For instance, a 2×2 matrix is dumped as follows.

```
f <-
structure(c(1, 2, 3, 4), .Dim = c(2L, 2L))
```

Here the dimensions are defined to be `c(2L, 2L)`. Stan always treats these dimesions as `L`.

Part III

Modeling Language Reference

7. Data Types

Every expression in Stan has a type that is uniquely determined at compile time. The basic Stan data types are real and integer, vector of real, row vector of real, and matrix of real.

Integer or real types may be constrained with lower bounds, upper bounds, or both. Vector data types may be constrained to increasing positive values or simplexes. Matrix data types may be constrained to correlation matrices or covariance matrices.

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.

7.1. Finite-Precision Arithmetic

Unfortunately, the lovely mathematical abstraction of integers and real numbers is only partially supported by finite-precision computer arithmetic.

Integers

Stan uses 64-bit (8-byte) integers for all of its integer representations. The maximum value that can be represented as an integer is $2^{63} - 1$; the minimum value is $-(2^{63})$.

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.

Reals

Stan uses 64-bit (8-byte) floating point representations of real numbers. Stan roughly[1] follows the IEEE 754 standard for floating-point computation. The range of a 64-bit number is roughly $\pm 2^{1022}$, which is slightly larger than $\pm 10^{307}$. It is a good idea to stay well away from such extreme values in Stan models as they are prone to cause overflow.

[1] Stan compiles integers to `long 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.

64-bit floating point representations have roughly 16 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. This will cause rounding errors for very large integers .

Real values are not demoted to integers. For examples, real values may only be assigned to real variables, but integer values may be assigned to either integer variables or real variables. What happens interally is that the integer representation is converted to a floating-point representation. This operation is not free and is thus best avoided if possible.

7.2. 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 secton 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 using

```
int N;
```

As in this example, all variable declarations must end with semicolon.

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

```
int(1,) N;
```

This illustrates that the bounds are inclusive for integers.

To declare an integer variable `cond` to take only binary values, that is 0 or 1, a lower and upper bound must be provided, as in

```
int(0,1) cond;
```

Unconstrained Real

Unconstrained real variables are declared using the keyword `real`. For example,

```
real theta;
```

declares a real valued variable `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 by

```
real(0,) sigma;
```

The variable `x` may be declared to be less than -1 by

```
real(,-1) x;
```

To ensure `rho` takes on values between -1 and 1, use

```
real(-1,1) rho;
```

Expressions as Bounds

Bounds for integer or real variables may be arbitrary expressions, the only requirement being that they do not include non-data variables. That is, any variable used in an expression declaring a bound must be declared in the data block or the transformed data block. For example, it is acceptable to have the following

```
data {
 real lb;
}
parameters {
 real(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 involve arbitrary expressions so long as the result is of type integer and the only variables involved are data or transformed data variables. For instance,

```
data {
 int(1,) N;
 real y[N];
}
parameters {
 real(min(y),max(y)) phi;
}
```

This declares a positive integer data variable `N`, an array `y` of real-valued data of length `N`, and then a parameter ranging between the minimum and maximum value of `y`.

7.3. Vector and Matrix Data Types

Indexing

Vectors and matrices, as well as arrays, are indexed starting from 1 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 from 0.

Unconstrained 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 in

```
vector(3) u;
```

Unit Simplices

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. Simplices are declared with their full dimensionality. For instance, `theta` is declared to be a unit 5-simplex by

```
simplex(5) theta;
```

Unit simplices are implemented as vectors and may be assigned to other vectors.

Positive, Ordered Vectors

A positive ordered vector is a vector whose positive entries are sorted in ascending order. For instance, $(1.0, 2.7, 2.71)^\top$ is a positive, ordered vector. Positive ordered vectors are most often employed as cut points in ordinal logistic regression models.

The variable `c` is declared as an ordered 5-vector of positive values by

```
pos_ordered(5) c;
```

After their declaration, positive, ordered vectors, like unit simplices, may be assigned to other vectors and other vectors may be assigned to them.

Unconstrained 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;
```

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 done element-wise in a loop or by using the transpose operator.

Unconstrained 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.

Correlation Matrices

Matrix variables may be constrained to represent correlation matrices. A matrix is a correlation matrix if it is positive definite (and hence square and symmetric), has entries between -1 and 1, and has a unit diagonal. Because correlation matrices are square, they only need one dimension 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 positive definite (and hence square and symmetric with positive diagonal entries). 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`.

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) a;
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]

7.4. 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 5 integers by

```
int n[5];
```

A 2-dimensional array of real values with 3 rows and 4 columns is delcared with

```
real a[3,4];
```

A 3-dimensional array `z` of positive reals with 5 rows, 4 columns, and 2 shelfs is declared by

```
real(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

[2] As of the beta version of Stan version 1.0, the form `m[2,3]` is more efficient because it does not require the creation and use of an intermediate expression template for `m[2]`. In later versions, explicit calls to `m[2][3]` may be optimized to be as efficient as `m[2,3]` by the Stan compiler.

```
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.

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 2-dimesions, `y[1,1]` is the first element and `y[3,4]` the last element. For a 3-dimensional array `z`, the first element is `z[1,1,1]`, and so on.

Slices of arrays may be accessed by providing fewer than the full number of indexes. For example, suppose `y` is a 2-dimensional array with 3 rows and 4 columns. Then `y[3]` is 1-dimensional array of length 4. 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.

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]
```

8. Expressions

Stan's modeling language is more procedural than what users may be familiar with from BUGS and JAGS, both of which are declarative. For instance, Stan statements are executed in the order they are written in model specifications and local variables may be reassigned as in a procedural programming language. Furthermore, variables must be declared before they are use.

Like most programming languages, Stan is defined syntactically in terms of expressions, which denote values, and statements, which denote an action to be taken.

The purpose of Stan's modeling language is to allow users to write down a probability function up to a multiplicative normalizing constant.

8.1. Expressions

The top-level grammar for expressions is provided in Figure 8.1.

Valid Stan expresions include simple numerical literals (e.g., `2`, `32.7`), identifiers representing variables (e.g., `theta`),

The simplest form of expression is a literal denoting a value, such as `32.7`. Expressions may also consist of

```
expression ::= literal
             | variable
             | expression infixOp expression
             | prefixOp expression
             | expression postfixOp
             | expression '[' expressions ']'
             | function '(' expressions ')'
             | '(' expression ')'
```

Figure 8.1: *The top-level expression grammar for Stan model specifications.*

Part IV

Built-In Functions

9. Built-in Functions

Stan supports a wide range of built-in functions. These functions operate over and return values spanning the complete range of data types.

The following chapters in this part describe basic functions returning integers and returning real values, array and matrix functions, and probability functions and cumulative distributions.

9.1. Function Signatures

The documentation of the special functions begins with a function signature. This determines the type of arguments taken by the function and the type of value returned.

Overloading

Stan relies on function overloading, which means there can be several different functions with the same name. Functions are determined uniquely by the combination of their name and the type of their arguments.

For example, Stan includes two built-in functions named `max`, one that operates on integers to return an integer, with signature

```
int max(int m, int n)
```

and one that operates over reals, with signature

```
real max(real x, real y)
```

The appropriate instance is called based on the arguments provided.

The two functions named `max` are distinguished by their argument types. It would not be legal to add a third function with signature `real max(int,int)`, as it would lead to a conflict with the existing function of name `max` and argument types `(int,int)`, even though it has a different return type.

9.2. Integer-to-Real Type Promotion

Integer values may be passed to functions requiring real values. This is not unrelated to the fact that integer values may be assigned to real variables. In both cases, the underlying C++ code handles the promotion of the underlying integer type to the underlying floating-point type.

Consider again the two functions `max(int,int)` and `max(real,real)`, and suppose these are the only two functions named `max` (they are the only two of that name built into Stan). If `m` is an integer variable and `x` a real variable, the call `max(m,x)` must be resolved as referring to `max(double,double)`, because integers can be promoted to real values, but not vice-versa.

If `n` is another integer variable, a call to `max(m,n)` could conceivably call either function named `max`. The function `max(int,int)` requires no promotions, whereas `max(double,double)` requires two, so the first is invoked. The general rule is that the function call matches the function requiring the fewest promotions.

Even this simple disambiguation scheme leads to possible ambiguity (though Stan's built-in functions avoid it). If there were two functions, `foo(int,real)` and `foo(real,int)` and if `m` and `n` were integer variales, a call to `foo(m,n)` would be ambiguous. Either function could be matched with one promotion and there is no better-matching function. In such cases, the call will not compile. This could be resolved by assigning one variable to a temporary, as in the following example.

```
int m;  int n;  real x;  real y;
x <- m;  y <- n;
# foo(m,n);  // illegal!
foo(x,n);    // calls foo(double,int)
foo(m,y);    // calls foo(int,double)
```

10. Integer-Valued Basic Functions

This chapter describes Stan's built-in function that operate on real and integer arguments and return results of type integer.

10.1. Absolute Function

`int` **`abs`**`(int x)`

The absolute value of x

10.2. 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

11. Real-Valued Basic Functions

This chapter describes built-in functions that take zero or more real or integer arguments and return real values. Constants are represented as functions with no arguments and must be called as such.

11.1. Mathematical Constants

`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

11.2. Special Values

`real` **`nan`**`()`

Not-a-number, a special non-finite real value returned to signal an error

`real` **`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

11.3. Logical Functions

`real` **`if_else`**`(int cond, real x, real y)`

x if *cond* is non-zero, and *y* otherwise

`real` **`step`**`(real x)`

0 if *x* is negative and 1 otherwise

11.4. 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

11.5. 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*

11.6. Arithmetic Functions

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

The real value remainder after dividing *x* by *y*

11.7. 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

11.8. Power and Logarithm Functions

`real` **`sqrt`**`(real x)`

The square root of x

`real` **`cbrt`**`(real x)`

The cube root of x

`real` **`square`**`(real x)`

The square of x

`real` **`exp`**`(real x)`

The natural exponential of x

`real` **`exp2`**`(real x)`

The base-2 exponential of x

`real` **`expm1`**`(real x)`

The natural exponential of x minus 1

`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

11.9. Link Functions

`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

11.10. Trigonometric Functions

`real` **`hypot`**`(real x, real y)`

The length of the hypoteneuse 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 princpipal 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

11.11. 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

11.12. 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 normal density function of x

`real` **`log_loss`**`(int y, real y_hat)`

The log loss of predicting probabilty y_hat for binary outcome y

11.13. Combinatorial Functions

`real` **`tgamma`**`(real x)`

The gamma function applied to x

`real` **`lgamma`**`(real x)`

The natural log 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 log 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

11.14. Composed Functions

`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

12. Array Operations

13. Matrix Operations

14. Discrete Probability Functions

15. Continuous Probability Functions

16. Cumulative Distributions

Part V

Advanced Topics

17. Variable Transforms

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 any constraints in the parameter's definition. Constraints that may be placed on variables include upper and lower bounds, positive ordered vectors, simplex vectors, correlation matrices and covariance matrices. This chapter provides a definition of the transforms used for each type of variable.

Once the model is compiled, it has support on all of $\mathbf{R}^K$, where K is the number of unconstrained parameters needed to define the actual parameters defined in the model.

The details of section need not be understood in order to use Stan for well-behaved models. Understanding the sampling behavior of Stan fully requires understanding these transforms.

17.1. 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,

$$\text{support}(X) = \{x | p_X(x) > 0\}.$$

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 and the rest of the chapter details the transforms used by Stan.

Univariate Changes of Variables

Suppose X is one dimensional and $f : \text{support}(X) \to \mathbf{R}$ is a one-to-one, monotonic function with a differentiable inverse f^{-1}. Then the density of Y is given by

$$p_Y(y) = p_X(f^{-1}(y)) \left| \frac{d}{dy} f^{-1}(y) \right|.$$

Multivariate Changes of Variables

An absolute derivative measures how the scale of the transformed variable changes with respect to the underlying variable. The multivariate generalization of absolute

derivatives is the absolute Jacobian determinants. The Jacobian 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

$$p_Y(y) = p_X(g(y)) \, |\det \, J_g(y)\,|,$$

where $\det$ is the matrix determinant operation and $J_{f^{-1}}(y)$ is the Jacobian of f^{-1} evaluated at y. The latter is defined by

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

If the Jacobian is a triangular matrix, the determinant reduces to the product of the diagonal entries,

$$\det \, J_{f^{-1}}(y) = \prod_{k=1}^{K} \frac{\partial y_k}{\partial x_k}.$$

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 raw variable.

17.2. Lower Bounds

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 the lower-bound transform maps an unbounded variable Y to a variable X that is bounded below by a by

$$X = \exp(Y) + a.$$

Absolute Derivative of the Lower Bound Inverse Transform

The absolute derivative of the inverse transform is

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

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

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

17.3. Upper Bounds

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

$$Y = \log(b - X).$$

Inverse Upper Bound 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 Inverse Upper Bound Transform

The absolute derivative of the inverse upper bound transform is

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

Therefore, the density of the unconstrained variable Y is defined in terms of the density of the variable X with an upper bound of b by

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

17.4. Lower and Upper Bounds

For lower and upper-bounded variables, Stan uses a scaled and translated log-odds transform.

Log Odds and the Logistic Sigmoid

The log-odds function is defined for $u \in (0, 1)$ by

$$\text{logit}(u) = \log \frac{u}{1-u}.$$

The inverse of the log odds function is the logistic sigmoid, defined for $v \in (-\infty, \infty)$ by

$$\text{logit}^{-1}(v) = \frac{1}{1+\exp(-v)}.$$

The derivative of the logistic sigmoid is

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

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

$$Y = \text{logit}\left(\frac{X-a}{b-a}\right).$$

The inverse of this transform is

$$X = a + (b-a) \cdot \text{logit}^{-1}(Y)$$

and the density of the transformed variable is

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

Despite its apparent complexity, $\text{logit}^{-1}(y)$, and hence $\exp(-y)$, need only be evaluated once.

17.5. Positive Ordered

For some modeling tasks, a vector-valued random variable X is required with support on positive, ordered sequences. One example is the set of cut points in ordinal logistic regression.

In constraint terms, a positive, ordered vector $x \in \mathbf{R}^K$ is a vector that satisfies

$$0 < x_1$$

and for $2 \leq k \leq K$,

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

Stan's transform follows the constraint directly. It maps a vector $x \in \mathbf{R}^K$ to a vector $f(x) = y \in \mathbf{R}^K$ by setting

$$y_1 = f_1(x) = \log x_1$$

and for $2 \leq k \leq K$,

$$y_k = f_k(x) = \log\left(x_k - x_{k-1}\right).$$

The inverse transform $x = f^{-1}(y)$ satisfies

$$x_1 = \exp(y_1)$$

and for $2 \leq k \leq K$,

$$x_k = x_{k-1} + \exp(y_k) = \sum_{k'=1}^{k} \exp(y_{k'}).$$

The Jacobian of the inverse transform f^{-1} is lower triangular, with diagonal elements for $1 \leq k \leq K$ of

$$J_{k,k} = \frac{\partial}{\partial y_k} f_k^{-1}(y) = \exp(y_k).$$

Because of the triangularity and the positivivity of the x_k and their differences, the absolute determinant of the Jacobian is

$$|\det J| = \prod_{k=1}^{K} J_{k,k} = \prod_{k=1}^{K} \exp(y_k).$$

Putting this all together, if p_X is the density of X, then the transformed variable Y has density p_Y given by

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

17.6. Unit Simplex

The parameter of the K-dimensional categorical distribution must lie in the unit K-simplex. Consequently, simplex-constrained variables show up in multivariate discrete models of all kinds.

The K-simplex is the set of points $x \in \mathbf{R}^K$ 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 hull of the convex closure of the vertices. For instance, in 2-dimensions, the basis points are the extreme values $(0, 1)$, and $(1, 0)$ and the unit 2-simplex is interval with these as the end points. In 3-dimensions, the basis is $(0, 0, 1)$, $(0, 1, 0)$ and $(1, 0, 0)$ and the unit 3-simplex is the triangle with these vertices. As these examples illustrate, the simplex always picks out a subspace of $K-1$ dimensions from $\mathbf{R}^K$.

A point x in the K-simplex is fully determined by its first $K - 1$ dimensions, because rearranging terms in the constraint yields

$$x_K = 1 - \sum_{k=1}^{K-1} x_k.$$

Stan employs a transform whose inverse may be understood using a stick-breaking metaphor. A simplex is determined by taking a stick of unit length, breaking a piece off, the length of which is x_1. Then x_2 is determined by breaking a piece from what's left. A total of $K - 1$ pieces are broken off, determining $x_1, \ldots, x_{K-1}$. To complete the metaphor, the length of the remaining piece after $K - 1$ pieces are broken off determines x_K.

The simplex transform f is most easily understood in terms of its inverse $x = f^{-1}(y)$, which maps a point in $y \in \mathbf{R}^{K-1}$ to a point x in the K-simplex. An intermediate vector $z \in \mathbf{R}^{K-1}$, whose coordinates z_k represent the proportion of the stick broken off in step k, is defined elementwise for $1 \leq k < K$ by

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

The logit term in the above definition adjusts the transform so that a zero vector y is mapped to $(1/K, \ldots, 1/K)$. For instance, if $y_1 = 0$, then $z_1 = 1/K$; if $y_2 = 0$, then $z_2 = 1/(K - 1)$; and if $z_{K-1} = 0$, then $z_{K-1} = 1/2$. This ensures that random initializations for categorical distribution parameters are initialized around a parameter value when $y = 0$ representing the uniform distribution.

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 stick left at stage k. This is multiplied by the break proportion z_k to yield x_k. Because x lines in a K-simplex, x_K is determined from $x_1, \ldots, x_{K-1}$.

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

$$\frac{\partial z_k}{\partial y_K} = \frac{\partial}{\partial z_k} \text{logit}^{-1}\left(y_k - \text{logit}\left(\frac{1}{K - k + 1}\right)\right) = z_k(1 - z_k),$$

and

$$\frac{\partial x_k}{\partial z_k} = \left(1 - \sum_{k'=1}^{k-1} x_{k'}\right).$$

Note that the definition is recursive, defininingx x_k in terms of $x_1, \ldots, x_{k-1}$.

Because the Jacobian J of f^{-1} is lower triangular and positve, 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

$$p_Y(y) = p_X(f^{-1}(y)) \prod_{k=1}^{K-1} z_k\,(1-z_k)\,\left(1 - \sum_{k'=1}^{k-1} x_{k'}\right).$$

This formula looks more complicated than it is. It only involves a single exponential function evaluation involved (in the logistic sigmoid applied to y_k to produce z_k); everything else is just basic arithmetic and keeping track of the remaing stick length.

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

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

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,

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

17.7. Correlation Matrices

A correlation matirx is a symmetric, positive-definite matrix with a unit diagonal. To deal with this rather complicated constraint, Stan implements the transform of [**?**], henceforth the LKJ-transform. 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 this transform in reverse, 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 consists of $\binom{K}{2}$ unconstrained values. Next, define a $K \times K$ matrix z by first transforming y using the $\tanh$ function, then filling in the triangle above the diagonal in column-major order with the results,

$$z_{i,j} = \tanh y_{i+(j-1)(j-2)/2}.$$

The bijective function $\tanh : \mathbf{R} \to (0, 1)$ is defined by

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

For example, in the 4×4 case, there are $\binom{4}{2}$ non-zero 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 map the matrix 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 $z_{i,j}$ is just the Pearson correlation between i and j.

In Stan, the LKJ transform is reformulated in terms of a Cholesky factor w of the final correlation matrix, defined for $1 \leq i, j \leq K$ by

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

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

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

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 CPCs as

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

which is also the square of the determinant of the triangular w.

Jacobian Determinant of the Inverse Correlation Transform

Correlation 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,

$$w = \text{cholesky}(x).$$

The next step from the Cholesky factor w back to the matrix z of CPCs is simplified by the ordering of the elements in the definition of of w, which when inverted yields

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

XThe final stage of the transform reverses the tanh transform, defining

$$y_{j(j-1)/2+i} = \tanh^{-1} z_{i,j}$$

for $i < j$, where the inverse of tanh is given by

$$\tanh^{-1} v = \frac{1}{2} \log \left(\frac{1+v}{1-v} \right).$$

17.8. Covariance Matrices

Covariance matrices are just scaled correlation matrices. This requires an additional K positive scaling parameters, for a total requirement of $K + \binom{K}{2}$ parameters to specify a covariance matrix.

Covariance Matrix Inverse Transform

Suppose y is a $\binom{K}{2}$-dimensional array specifying the $K \times K$ correlation matrix x as specified by the correlation matrix inverse transform described in the previous section.

Let y' be a K-dimensional vector of unconstrained scaling parameters. An exponential transform converts these to positive values component-wise, by

$$u = \exp(y').$$

The covariance matrix is the scaled version of the correlation matrix x,

$$v \;=\; \text{diag}(u)\; x\; \text{diag}(u) \;=\; (\text{diag}(u)\; z)\; (\text{diag}(u)\; z))^\top,$$

where $\text{diag}(u)$ is the diagonal matrix with diagonal u.

18. The C++ Model Class

The generated C++ class extends a built-in Stan abstract base class for probability models. Instances of the class are constructed from a specified data vector y. The data vector y determines the dimensionality K of the parameter vector θ, which in general may depend on size constants in y. The class implements a method that takes a parameter K-vector θ as argument and returns the (unnormalized) total log probabilty,

$$\theta \mapsto \log p(y, \theta)$$

The second method returns the gradient of the (unnormalized) total probability as a function of a parameter K-vector θ,

$$\theta \mapsto \nabla_\theta \log p(y, \theta) = (\frac{\partial}{\partial \theta_1} \log p(y, \theta), \ldots, \frac{\partial}{\partial \theta_K} \log p(y, \theta)),$$

The class computes gradients using accurate and efficient reverse-mode algorithmic differentiaton. The cost of computing the gradient is a small multiple of the cost of computing the log probability. The cost inovlves a bounded amount of extra bookkeeping for each subexpression involved in computing the log probability. Unlike in the calculation of finite differences, the extra bookkeeping is not dependent on the dimensionality of the parameter vector.

19. Optimizing Stan Code

Appendices

A. Installation

References