Stan Reference Manual

Version 2.35

Stan Development Team

Table of Contents

Overview 1

I Language **2**

1. Character Encoding 4
- 1.1 Content characters 4
- 1.2 Comment characters 4
- 1.3 String literals 4

2. Includes 5
- 2.1 Recursive includes 6
- 2.2 Include paths 7

3. Comments 8
- 3.1 Line-based comments 8
- 3.2 Bracketed comments 8

4. Whitespace 9
- 4.1 Whitespace characters 9
- 4.2 Whitespace neutrality 9
- 4.3 Whitespace location 9

5. Data Types and Declarations 10
- 5.1 Overview of data types 10
- 5.2 Primitive numerical data types 12
- 5.3 Complex numerical data type 14
- 5.4 Scalar data types and variable declarations 15
- 5.5 Vector and matrix data types 20
- 5.6 Array data types 27
- 5.7 Tuple data type 32
- 5.8 Variable types vs. constraints and sizes 35

5.9 Variable declaration 36
5.10 Compound variable declaration and definition 40
5.11 Declaring multiple variables at once 41

6. Expressions 43
6.1 Numeric literals 43
6.2 Variables 44
6.3 Container expressions 47
6.4 Parentheses for grouping 50
6.5 Arithmetic and matrix operations on expressions 50
6.6 Conditional operator 53
6.7 Indexing 55
6.8 Multiple indexing and range indexing 56
6.9 Function application 58
6.10 Type inference 60
6.11 Higher-order functions 64
6.12 Chain rule and derivatives 67

7. Statements 69
7.1 Statement block contexts 69
7.2 Assignment statements 69
7.3 Increment log density 73
7.4 Sampling statements 75
7.5 Distribution statements 75
7.6 For loops 85
7.7 Foreach loops 86
7.8 Conditional statements 87
7.9 While statements 88
7.10 Statement blocks and local variable declarations 89
7.11 Break and continue statements 91
7.12 Print statements 93
7.13 Reject statements 96
7.14 Fatal error statements 98

8. Program Blocks 99

8.1 Overview of Stan's program blocks 99
8.2 Statistical variable taxonomy 103
8.3 Program block: `data` 105
8.4 Program block: `transformed data` 106
8.5 Program block: `parameters` 106
8.6 Program block: `transformed parameters` 108
8.7 Program block: `model` 109
8.8 Program block: `generated quantities` 109

9. User-Defined Functions 111
9.1 Function-definition block 111
9.2 Function names 111
9.3 Calling functions 112
9.4 Argument types and qualifiers 114
9.5 Function bodies 115
9.6 Parameters are constant 117
9.7 Return value 118
9.8 Void Functions as Statements 119
9.9 Declarations 120

10. Constraint Transforms 121
10.1 Limitations due to finite accuracy presentation 121
10.2 Changes of variables 122
10.3 Lower bounded scalar 123
10.4 Upper bounded scalar 124
10.5 Lower and upper bounded scalar 125
10.6 Affinely transformed scalar 126
10.7 Ordered vector 127
10.8 Unit simplex 128
10.9 Unit vector 131
10.10 Correlation matrices 132
10.11 Covariance matrices 135
10.12 Cholesky factors of covariance matrices 138
10.13 Cholesky factors of correlation matrices 139

11. Language Syntax 142
11.1 BNF grammars 142
11.2 Tokenizing rules 148
11.3 Extra-grammatical constraints 149

12. Program Execution 152
12.1 Reading and transforming data 152
12.2 Initialization 153
12.3 Sampling 154
12.4 Optimization 156
12.5 Variational inference 156
12.6 Model diagnostics 156
12.7 Output 157

13. Deprecated Features 158
13.1 `lkj_cov` distribution 158
13.2 New Keywords 159
13.3 Deprecated Functions 159

14. Removed Features 160
14.1 `lp__` variable 160
14.2 Assignment with `<-` 160
14.3 `increment_log_prob` statement 160
14.4 `get_lp()` function 161
14.5 `_log` density and mass functions 161
14.6 `cdf_log` and `ccdf_log` cumulative distribution functions 161
14.7 User-defined function with `_log` suffix 161
14.8 `if_else` function 162
14.9 Character # as comment prefix 162
14.10 Postfix brackets array syntax 162
14.11 Nested multiple indexing in assignments 163
14.12 Real values in conditionals 163

II Algorithms **165**

15. MCMC Sampling 167
15.1 Hamiltonian Monte Carlo 167
15.2 HMC algorithm parameters 170
15.3 Sampling without parameters 176
15.4 General configuration options 177
15.5 Divergent transitions 179

16. Posterior Analysis 181
16.1 Markov chains 181
16.2 Convergence 182
16.3 Notation for samples, chains, and draws 182
16.4 Effective sample size 186

17. Optimization 191
17.1 General configuration 191
17.2 BFGS and L-BFGS configuration 191
17.3 Writing models for optimization 193

18. Pathfinder 194

19. Variational Inference 195
19.1 Stochastic gradient ascent 195

20. Laplace Approximation 197

21. Diagnostic Mode 198
21.1 Diagnostic mode output 198
21.2 Configuration options 199
21.3 Speed warning and data trimming 199

III Usage **200**

22. Reproducibility 202
22.1 Notable changes across versions 203

23. Licenses and Dependencies 204
23.1 Stan license 204
23.2 Boost license 204
23.3 Eigen license 204
23.4 SUNDIALS license 205
23.5 Threaded Building Blocks (TBB) License 205
23.6 Google test license 205

References 206

Overview

This is the official reference manual for Stan's *programming language* for coding probability models, *inference algorithms* for fitting models and making predictions, and *posterior analysis* tools for evaluating the results. This manual applies to all Stan interfaces.

The first part of the reference manual provides a full specification of the Stan programming language. The language is responsible for defining a log density function conditioned on data. Typically, this is a Bayesian posterior, but it may also be a penalized likelihood function. The second part of the manual specifies the inference algorithms and posterior inference tools. The third part provides auxiliary information about the use of Stan.

Part I

Language

1. Character Encoding

1.1. Content characters

The content of a Stan program must be coded in ASCII. All identifiers must consist of only ASCII alpha-numeric characters and the underscore character. All arithmetic operators and punctuation must be coded in ASCII.

Compatibility with Latin-1 and UTF-8

The UTF-8 encoding of Unicode and the Latin-1 (ISO-8859-1) encoding share the first 128 code points with ASCII and thus cannot be distinguished from ASCII. That means you can set editors, etc., to use UTF-8 or Latin-1 (or the other Latin-n variants) without worrying that the content of a Stan program will be destroyed.

1.2. Comment characters

Any bytes on a line after a line-comment sequence (`//` or `#`) are ignored up until the ASCII newline character (`\n`). They may thus be written in any character encoding which is convenient.

Any content after a block comment open sequence in ASCII (`/*`) up to the closing block comment (`*/`) is ignored, and thus may also be written in whatever character set is convenient.

1.3. String literals

The raw byte sequence within a string literal is escaped according to the C++ standard. In particular, this means that UTF-8 encoded strings are supported, however they are not tested for invalid byte sequences. A `print`, `reject`, or `fatal_error` statement should properly display Unicode characters if your terminal supports the encoding used in the input. In other words, Stan simply preserves any string of bytes between two double quotes (`"`) when passing to C++. On compliant terminals, this allows the use of glyphs and other characters from encodings such as UTF-8 that fall outside the ASCII-compatible range.

ASCII is the recommended encoding for maximum portability, because it encodes the ASCII characters (Unicode code points 0–127) using the same sequence of bytes as the UTF-8 encoding of Unicode and common ISO-8859 extensions of Latin.

2. Includes

Stan allows one file to be included within another file using a syntax similar to that from C++. For example, suppose the file `my-std-normal.stan` defines the standard normal log probability density function (up to an additive constant).

```
functions {
  real my_std_normal_lpdf(vector y) {
    return -0.5 * y' * y;
  }
}
```

Suppose we also have a file containing a Stan program with an include statement.

```
#include my-std-normal.stan
parameters {
  real y;
}
model {
  y ~ my_std_normal();
}
```

This Stan program behaves as if the contents of the file `my-std-normal.stan` replace the line with the `#include` statement, behaving as if a single Stan program were provided.

```
functions {
  real my_std_normal_lpdf(vector y) {
    return -0.5 * y' * y;
  }
}
parameters {
  real y;
}
model {
  y ~ my_std_normal();
}
```

There are no restrictions on where include statements may be placed within a file or

what the contents are of the replaced file.

Space before includes

It is possible to use includes on a line non-initially. For example, the previous example could've included space before the # in the include line:

```
    #include my-std-normal.stan
parameters {
// ...
```

If there is initial space before an include, it will be discarded.

Comments after includes

It is also possible to include line-based comments after the include. For example, the previous example can be coded as:

```
#include my-std-normal.stan  // definition of standard normal
parameters {
// ...
```

Line comments are discarded when the entire line is replaced with the contents of the included file.

2.1. Recursive includes

Recursive includes will lead to a compiler error. For example, suppose `a.stan` contains

```
#include b.stan
```

and `b.stan` contains

```
#include a.stan
```

This will result in an error explaining the circular dependency:

```
Syntax error in './b.stan', line 1, column 0, included from
'./a.stan', line 1, column 0, included from
'./b.stan', line 1, column 0, included from
'a.stan', line 1, column 0, include error:
   -------------------------------------------------
     1:  #include a.stan
         ^
   -------------------------------------------------

File a.stan recursively included itself.
```

2.2. Include paths

The Stan interfaces may provide a mechanism for specifying a sequence of system paths in which to search for include files. The file included is the first one that is found in the sequence.

Slashes in include paths

If there is not a final / or \ in the path, a / will be appended between the path and the included file name.

3. Comments

Stan supports C++-style line-based and bracketed comments. Comments may be used anywhere whitespace is allowed in a Stan program.

3.1. Line-based comments

Any characters on a line following two forward slashes (`//`) is ignored along with the slashes. These may be used, for example, to document variables,

```
data {
  int<lower=0> N;  // number of observations
  array[N] real y;  // observations
}
```

3.2. Bracketed comments

For bracketed comments, any text between a forward-slash and asterisk pair (`/*`) and an asterisk and forward-slash pair (`*/`) is ignored.

4. Whitespace

4.1. Whitespace characters

The whitespace characters (and their ASCII code points) are the space (0x20), tab (0x09), carriage return (0x0D), and line feed (0x0A).

4.2. Whitespace neutrality

Stan treats all whitespace characters identically. Specifically, there is no significance to indentation, to tabs, to carriage returns or line feeds, or to any vertical alignment of text. Any whitespace character is exchangeable with any other.

Other than for readability, the number of whitespaces is also irrelevant. One or more whitespace characters of any type are treated identically by the parser.

4.3. Whitespace location

Zero or more whitespace characters may be placed between symbols in a Stan program. For example, zero or more whitespace characters of any variety may be included before and after a binary operation such as `a * b`, before a statement-ending semicolon, around parentheses or brackets, before or after commas separating function arguments, etc.

Identifiers and literals may not be separated by whitespace. Thus it is not legal to write the number `10000` as `10 000` or to write the identifier `normal_lpdf` as `normal _ lpdf`.

5. Data Types and 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.

1. 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.
2. 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 (called through an interface such as CmdStan, RStan, or PyStan) will flag any type errors and indicate the offending expressions quickly when the program is compiled.
3. 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.

5.1. Overview of data types

Arguments for built-in and user-defined functions and local variables are required to be basic data types, meaning an unconstrained scalar, vector, or matrix type, or an array of such.

Passing arguments to functions in Stan works just like assignment to basic types. Stan functions are only specified for the basic data types of their arguments, including array dimensionality, but not for sizes or constraints. Of course, functions often check constraints as part of their behavior.

Primitive types

Stan provides two primitive data types, `real` for continuous values and `int` for integer values. These are both considered scalar types.

Complex types

Stan provides a complex number data type `complex`, where a complex number contains both a real and an imaginary component, both of which are of type `real`. Complex types are considered scalar types.

Vector and matrix types

Stan provides three real-valued matrix data types, `vector` for column vectors, `row_vector` for row vectors, and `matrix` for matrices.

Stan also provides three complex-valued matrix data types, `complex_vector` for column vectors, `complex_row_vector` for row vectors, and `complex_matrix` for matrices.

Array types

Any type (including the constrained types discussed in the next section) can be made into an array type by declaring array arguments. For example,

```
array[10] real x;
array[6, 7] matrix[3, 3] m;
array[12, 8, 15] complex z;
```

declares `x` to be a one-dimensional array of size 10 containing real values, declares `m` to be a two-dimensional array of size 6×7 containing values that are 3×3 matrices, and declares `z` to be a $12 \times 8 \times 15$ array of complex numbers.

Prior to 2.26 Stan models used a different syntax which has since been removed. See the Removed Features chapter for more details.

Tuple types

For any sequence of types, Stan provides a tuple data type. For example,

```
tuple(real, array[5] int) xi;
```

declares `xi` to be a tuple holding two values, the first of which is of type type `real` and the second of which a 5-dimensional array of type `int`.

Constrained data types

Declarations of variables other than local variables may be provided with constraints. These constraints are not part of the underlying data type for a variable,

but determine error checking in the transformed data, transformed parameter, and generated quantities block, and the transform from unconstrained to constrained space in the parameters block.

All of the basic data types other than `complex` may be given lower and upper bounds using syntax such as

```
int<lower=1> N;
real<upper=0> log_p;
vector<lower=-1, upper=1>[3] rho;
```

There are also special data types for structured vectors and matrices. There are four constrained vector data types, `simplex` for unit simplexes, `unit_vector` for unit-length vectors, `ordered` for ordered vectors of scalars and `positive_ordered` for vectors of positive ordered scalars. There are specialized matrix data types `corr_matrix` and `cov_matrix` for correlation matrices (symmetric, positive definite, unit diagonal) and covariance matrices (symmetric, positive definite). The type `cholesky_factor_cov` is for Cholesky factors of covariance matrices (lower triangular, positive diagonal, product with own transpose is a covariance matrix). The type `cholesky_factor_corr` is for Cholesky factors of correlation matrices (lower triangular, positive diagonal, unit-length rows).

Constraints provide error checking for variables defined in the `data`, `transformed data`, `transformed parameters`, and `generated quantities` blocks. Constraints are critical for variables declared in the `parameters` block, where they determine the transformation from constrained variables (those satisfying the declared constraint) to unconstrained variables (those ranging over all of $\mathbb{R}^n$).

It is worth calling out the most important aspect of constrained data types:

The model must have support (non-zero density, equivalently finite log density) at parameter values that satisfy the declared constraints.

If this condition is violated with parameter values that satisfy declared constraints but do not have finite log density, then the samplers and optimizers may have any of a number of pathologies including just getting stuck, failure to initialize, excessive Metropolis rejection, or biased draws due to inability to explore the tails of the distribution.

5.2. 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 value is determined by the underlying architecture. On most, their values wrap, but this cannot be guaranteed. 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 truncates the result of division (see the Stan Functions Reference integer-valued arithmetic operators section for more information).

Reals

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

64-bit floating point representations have roughly 15 decimal digits of accuracy. But when they are combined, the result often has less accuracy. In some cases, the difference in accuracy between two operands and their result is large.

There are three special real values used to represent (1) not-a-number value for error conditions, (2) positive infinity for overflow, and (3) negative infinity for overflow. The behavior of these special numbers follows standard IEEE 754 behavior.

Not-a-number

The not-a-number value propagates. If an argument to a real-valued function is not-a-number, it either rejects (an exception in the underlying C++) or returns not-a-number itself. For boolean-valued comparison operators, if one of the arguments is not-a-number, the return value is always zero (i.e., false).

Infinite values

Positive infinity is greater than all numbers other than itself and not-a-number; negative infinity is similarly smaller. Adding an infinite value to a finite value returns the infinite value. Dividing a finite number by an infinite value returns zero; dividing an infinite number by a finite number returns the infinite number

[1] Stan compiles integers to `int` and reals to `double` types in C++. Precise details of rounding will depend on the compiler and hardware architecture on which the code is run.

of appropriate sign. Dividing a finite number by zero returns positive infinity. Dividing two infinite numbers produces a not-a-number value as does subtracting two infinite numbers. Some functions are sensitive to infinite values; for example, the exponential function returns zero if given negative infinity and positive infinity if given positive infinity. Often the gradients will break down when values are infinite, making these boundary conditions less useful than they may appear at first.

Promoting integers to reals

Stan automatically promotes integer values to real values if necessary, but does not automatically demote real values to integers. For very large integers, this will cause a rounding error to fewer significant digits in the floating point representation than in the integer representation.

Unlike in C++, real values are never demoted to integers. Therefore, real values may only be assigned to real variables. Integer values may be assigned to either integer variables or real variables. Internally, the integer representation is cast to a floating-point representation. This operation is not without overhead and should thus be avoided where possible.

5.3. Complex numerical data type

The `complex` data type is a scalar, but unlike `real` and `int` types, it contains two components, a real and imaginary component, both of which are of type `real`. That is, the real and imaginary components of a complex number are 64-bit, IEEE 754-complaint floating point numbers.

Constructing and accessing complex numbers

Imaginary literals are written in mathematical notation using a numeral followed by the suffix `i`. For example, the following example constructs a complex number $2 - 1.3i$ and assigns it to the variable `z`.

```
complex z = 2 - 1.3i;
real re = get_real(z);  // re has value 2.0
real im = get_imag(z);  // im has value -1.3
```

The getter functions then extract the real and imaginary components of `z` and assign them to `re` and `im` respectively.

The function `to_complex` constructs a complex number from its real and imaginary components. The functional form needs to be used whenever the components are not literal numerals, as in the following example.

```
vector[K] re;
vector[K] im;
// ...
for (k in 1:K) {
  complex z = to_complex(re[k], im[k]);
  // ...
}
```

Promoting real to complex

Expressions of type `real` may be assigned to variables of type `complex`. For example, the following is a valid sequence of Stan statements.

```
real x = 5.0;
complex z = x;  // get_real(z) == 5.0, get_imag(z) == 0
```

The real number assigned to a complex number determine's the complex number's real component, with the imaginary component set to zero.

Assignability is transitive, so that expressions of type `int` may also be assigned to variables of type `complex`, as in the following example.

```
int n = 2;
complex z = n;
```

Function arguments also support promotion of integer or real typed expressions to type `complex`.

5.4. Scalar 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 scalar types, namely integer, real, and complex. The next section covers 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;
```

Unconstrained complex

Unconstrained complex numbers are declared using the keyword `complex`. The following example declares `z` to be an unconstrained complex variable.

```
complex z;
```

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 supplied as

data.

Affinely transformed real

Real variables may be declared on a space that has been transformed using an affine transformation $x \mapsto \mu + \sigma * x$ with offset μ and (positive) multiplier σ, using a syntax similar to that for bounds. While these transforms do not change the asymptotic sampling behaviour of the resulting Stan program (in a sense, the model the program implements), they can be useful for making the sampling process more efficient by transforming the geometry of the problem to a more natural multiplier and to a more natural offset for the sampling process, for instance by facilitating a non-centered parameterisation. While these affine transformation declarations do not impose a hard constraint on variables, they behave like the bounds constraints in many ways and could perhaps be viewed as acting as a sort of soft constraint.

The variable x may be declared to have offset 1 as follows.

```
real<offset=1> x;
```

Similarly, it can be declared to have multiplier 2 as follows.

```
real<multiplier=2> x;
```

Finally, we can combine both declarations to declare a variable with offset 1 and multiplier 2.

```
real<offset=1, multiplier=2> x;
```

As an example, we can give x a normal distribution with non-centered parameterization as follows.

```
parameters {
  real<offset=mu, multiplier=sigma> x;
}
model {
  x ~ normal(mu, sigma);
}
```

Recall that the centered parameterization is achieved with the code

```
parameters {
  real x;
}
model {
  x ~ normal(mu, sigma);
```

```
}
```

or equivalently

```
parameters {
  real<offset=0, multiplier=1> x;
}
model {
  x ~ normal(mu, sigma);
}
```

Expressions as bounds and offset/multiplier

Bounds (and offset and multiplier) for integer or real variables may be arbitrary expressions. The only requirement is that they only include variables that have been declared (though not necessarily defined) before the declaration. array[N] row_vector[D] x; 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 arbitrary 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;
   array[N] real y;
}
parameters {
   real<lower=min(y), upper=max(y)> phi;
}
```

This declares a positive integer data variable `N`, an array `y` of real-valued data of length `N`, and then a parameter ranging between the minimum and maximum value of `y`. As shown in the example code, the functions `min()` and `max()` may be applied to containers such as arrays.

A more subtle case involves declarations of parameters or transformed parameters based on parameters declared previously. For example, the following program will work as intended.

```
parameters {
  real a;
  real<lower=a> b;  // enforces a < b
}
transformed parameters {
  real c;
  real<lower=c> d;
  c = a;
  d = b;
}
```

The parameters instance works because all parameters are defined externally before the block is executed. The transformed parameters case works even though `c` isn't defined at the point it is used, because constraints on transformed parameters are only validated at the end of the block. Data variables work like parameter variables, whereas transformed data and generated quantity variables work like transformed parameter variables.

Declaring optional variables

A variable may be declared with a size that depends on a boolean constant. For example, consider the definition of `alpha` in the following program fragment.

```
data {
  int<lower=0, upper=1> include_alpha;
  // ...
}
parameters {
  vector[include_alpha ? N : 0] alpha;
  // ...
}
```

If `include_alpha` is true, the model will include the vector `alpha`; if the flag is false, the model will not include `alpha` (technically, it will include `alpha` of size 0,

which means it won't contain any values and won't be included in any output).

This technique is not just useful for containers. If the value of `N` is set to 1, then the vector `alpha` will contain a single element and thus `alpha[1]` behaves like an optional scalar, the existence of which is controlled by `include_alpha`.

This coding pattern allows a single Stan program to define different models based on the data provided as input. This strategy is used extensively in the implementation of the RStanArm package.

5.5. Vector and matrix data types

Stan provides three types of container objects: arrays, vectors, and matrices. Vectors and matrices are more limited kinds of data structures than arrays. Vectors are intrinsically one-dimensional collections of real or complex values, whereas matrices are intrinsically two dimensional. Vectors, matrices, and arrays are not assignable to one another, even if their dimensions are identical. A 3×4 matrix is a different kind of object in Stan than a 3×4 array.

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

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

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

For constructing vectors and matrices in Stan, see Vector, Matrix, and Array Expressions.

Indexing from 1

Vectors and matrices, as well as arrays, are indexed starting from one (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 arrays starting from zero.

Vectors

Vectors in Stan are column vectors; see below for information on row vectors. Vectors are declared with a size (i.e., a dimensionality). For example, a 3-dimensional real 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;
```

Similarly, they may be declared with a offset and/or multiplier, as in the following example

```
vector<offset=42, multiplier=3>[3] u;
```

Complex vectors

Like real vectors, complex vectors are column vectors and are declared with a size. For example, a 3-dimensional complex vector is declared with the keyword `complex_vector`, as follows.

```
complex_vector[3] v;
```

Complex vector declarations do not support any constraints.

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.

In high dimensional problems, simplexes may require smaller step sizes in the inference algorithms in order to remain stable; this can be achieved through higher target acceptance rates for samplers and longer warmup periods, tighter tolerances for optimization with more iterations, and in either case, with less dispersed parameter initialization or custom initialization if there are informative priors for some parameters.

Unit vectors

A unit vector is a vector with a norm of one. For instance, $[0.5, 0.5, 0.5, 0.5]^\top$ is a unit 4-vector. Unit vectors are sometimes used in directional statistics. Unit vectors are declared with their full dimensionality. For instance, `theta` is declared to be a unit 5-vector by

```
unit_vector[5] theta;
```

Unit vectors are implemented as vectors and may be assigned to other vectors and vice-versa. Unit vector variables, like other constrained variables, are validated to ensure that they are indeed unit length; for unit vectors, this is only done up to a statically specified accuracy threshold ϵ to account for errors arising from floating-point imprecision.

Ordered vectors

An ordered vector type in Stan represents a vector whose entries are sorted in ascending order. For instance, $(-1.3, 2.7, 2.71)^\top$ is an ordered 3-vector. Ordered vectors are most often employed as cut points in ordered logistic regression models (see section).

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 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.

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;
```

Offset and multiplier are also similar as for the following 3-row-vector with offset -42 and multiplier 3.

```
row_vector<offset=-42, multiplier=3>[3] 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.

Complex row vectors

Complex row vectors are declared with the keyword `complex_row_vector` and given a size in basic declarations. For example, a 12-dimensional complex row vector `v` would be declared as

```
complex_row_vector[12] v;
```

Complex row vectors do not allow constraints.

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;
```

Similarly, matrices can be declared to have a set offset and/or multiplier, as in this matrix with multiplier 5.

```
matrix<multiplier=5>[3, 4] B;
```

Assigning to rows of a matrix

Rows of a matrix can be assigned by indexing the left-hand side of an assignment statement. For example, this is possible.

```
matrix[M, N] a;
row_vector[N] b;
// ...
a[1] = b;
```

This copies the values from row vector `b` to `a[1]`, which is the first row of the matrix `a`. If the number of columns in `a` is not the same as the size of `b`, a run-time error is raised; the number of columns of `a` is `N`, which is also the number of columns of `b`.

Assignment works by copying values in Stan. That means any subsequent assignment to `a[1]` does not affect `b`, nor does an assignment to `b` affect `a`.

Complex matrices

Complex matrices are declared with the keyword `complex_matrix` and a number of rows and columns. For example,

```
complex_matrix[3, 3] C;
```

Complex matrices do not allow constraints.

Covariance matrices

Matrix variables may be constrained to represent covariance matrices. A matrix is a covariance matrix if it is symmetric and positive definite. Like correlation matrices, covariance matrices only need a single dimension in their declaration. For instance,

```
cov_matrix[K] Omega;
```

declares `Omega` to be a $K \times K$ covariance matrix, where K is the value of the data variable `K`.

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.

Cholesky factors of covariance matrices

Matrix variables may be constrained to represent the Cholesky factors of a covariance matrix. This is often more convenient or more efficient than representing covariance matrices directly.

A Cholesky factor L is an $M \times N$ lower-triangular matrix (if $m < n$ then $L[m, n] = 0$) with a strictly positive diagonal ($L[k, k] > 0$) and $M \geq N$. If L is a Cholesky factor, then $\Sigma = L\,L^\top$ is a covariance matrix (i.e., it is positive definite). The mapping between positive definite matrices and their Cholesky factors is bijective—every covariance matrix has a unique Cholesky factorization.

The typical case of a square Cholesky factor may be declared with a single dimension,

```
cholesky_factor_cov[4] L;
```

Cholesky factors of positive semi-definite matrices

In general, two dimensions may be declared, with the above being equal to `cholesky_factor_cov[4, 4]`. The type `cholesky_factor_cov[M, N]` may be used for the general $M \times N$ case to produce positive semi-definite matrices of rank M.

Cholesky factors of correlation matrices

Matrix variables may be constrained to represent the Cholesky factors of a correlation matrix.

A Cholesky factor for a correlation matrix L is a $K \times K$ lower-triangular matrix with positive diagonal entries and rows that are of length 1 (i.e., $\sum_{n=1}^{K} L_{m,n}^2 = 1$). If L is a Cholesky factor for a correlation matrix, then $L\,L^\top$ is a correlation matrix (i.e., symmetric positive definite with a unit diagonal).

To declare the variable `L` to be a `K` by `K` Cholesky factor of a correlation matrix, the following code may be used.

```
cholesky_factor_corr[K] L;
```

Assigning constrained variables

Constrained variables of all types may be assigned to other variables of the same unconstrained type and vice-versa. Matching is interpreted strictly as having the same basic type and number of array dimensions. Constraints are not considered, but basic data types are. For instance, a variable declared to be `real<lower=0, upper=1>` could be assigned to a variable declared as `real` and vice-versa. Similarly, a variable declared as `matrix[3, 3]` may be assigned to a variable declared as `cov_matrix[3]` or `cholesky_factor_cov[3]`, and vice-versa.

Checks are carried out at the end of each relevant block of statements to ensure constraints are enforced. This includes run-time size checks. The Stan compiler isn't able to catch the fact that an attempt may be made to assign a matrix of one dimensionality to a matrix of mismatching dimensionality.

Promoting real to complex matrixes

Real-valued vectors, row vectors and matrices may be assigned to complex-valued vectors, row vectors and matrices, respectively. For example, the following is legal.

```
vector[N] v = ...;
complex_vector[N] u = 2 * v;
```

Row vectors and matrices work the same way.

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, 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.

Complex versions work the same way,

```
complex_matrix[M, N] m = ...;
complex_row_vector[N] u = m[3];
complex_vector[M] v = m[ , 2];
```

Array index style

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.

Size declaration restrictions

An integer expression is used to pick out the sizes of vectors, matrices, and arrays. For instance, we can declare a vector of size `M + N` using

```
vector[M + N] y;
```

Any integer-denoting expression may be used for the size declaration, providing all variables involved are either data, transformed data, or local variables. That is, expressions used for size declarations may not include parameters or transformed parameters or generated quantities.

5.6. Array data types

Stan supports arrays of arbitrary dimension. The values in an array can be any type, so that arrays may contain values that are simple reals or integers, vectors, matrices, or other arrays. Arrays are the only way to store sequences of integers, and some functions in Stan, such as discrete distributions, require integer arguments.

A two-dimensional array is just an array of arrays, both conceptually and in terms

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

Vectors, matrices, and arrays are not assignable to one another, even if their dimensions are identical.

For constructing arrays in Stan, see Vector, Matrix, and Array Expressions.

Declaring array variables

Arrays are declared with the keyword `array` followed by the dimensions enclosed in square brackets, the element type, and the name of the variable.

The variable `n` is declared as an array of five integers as follows.

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

A two-dimensional array of complex values with three rows and four columns is declared as follows.

```
array[3, 4] complex a;
```

A three-dimensional array `z` of positive reals with five rows, four columns, and two shelves can be declared as follows.

```
array[5, 4, 2] real<lower=0> z;
```

Arrays may also be declared to contain vectors. For example,

```
array[3] vector[7] mu;
```

declares `mu` to be an array of size 3 containing vectors with 7 elements. Arrays may also contain matrices. The example

```
array[15, 12] complex_matrix[7, 2] mu;
```

declares a 15 by 12 array of 7×2 complex matrices. Any of the constrained types may also be used in arrays, as in the declaration

```
array[2, 3, 4] cholesky_factor_cov[5, 6] mu;
```

of a $2 \times 3 \times 4$ array of 5×6 Cholesky factors of covariance matrices.

Accessing array elements and subarrays

If x is a 1-dimensional array of length 5, then x[1] is the first element in the array and x[5] is the last. For a 3×4 array y of two dimensions, y[1, 1] is the first element and y[3, 4] the last element. For a three-dimensional array z, the first element is z[1, 1, 1], and so on.

Subarrays of arrays may be accessed by providing fewer than the full number of indexes. For example, suppose y is a two-dimensional array with three rows and four columns. Then y[3] is one-dimensional array of length four. This means that y[3][1] may be used instead of y[3, 1] to access the value of the first column of the third row of y. The form y[3, 1] is the preferred form (see note in this chapter).

Assigning

Subarrays may be manipulated and assigned just like any other variables. Similar to the behavior of matrices, Stan allows blocks such as

```
array[9, 10, 11] real w;
array[10, 11] real x;
array[11] real y;
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]
```

Complex-valued arrays work the same way.

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.

```
array[3, 4] vector[5] a;
array[4] vector[5] b;
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.

```
array[3, 4] matrix[6, 5] d;
array[4] matrix[6, 5] e;
matrix[6, 5] f;
row_vector[5] g;
real x;
```

With these declarations, the following definitions are legal.

```
e = d[1];           // result is array of matrices
f = d[1, 3];        // result is matrix
f = e[3];           //   same result as above
g = d[1, 3, 2];     // result is row vector
g = e[3, 2];        //   same result as above
g = f[2];           //   same result as above
x = d[1, 3, 5, 2]; // result is scalar
x = e[3, 5, 2];     //   same result as above
x = f[5, 2];        //   same result as above
x = g[2];           //   same result as above
```

As shown, the result `f[2]` of supplying a single index to a matrix is the indexed row, here row 2 of matrix `f`.

Partial array assignment

Subarrays of arrays may be assigned by indexing on the left-hand side of an assignment statement. For example, the following is legal.

```
array[I, J, K] real x;
array[J, K] real y;
array[K] real z;
// ...
x[1] = y;
x[1, 1] = z;
```

The sizes must match. Here, `x[1]` is a `J` by `K` array, as is `y`.

Partial array assignment also works for arrays of matrices, vectors, and row vectors.

Mixing array, vector, and matrix types

Arrays, row vectors, column vectors and matrices are not interchangeable in Stan. Thus a variable of any one of these fundamental types is not assignable to any of the others, nor may it be used as an argument where the other is required (use as arguments follows the assignment rules).

Mixing vectors and arrays

For example, vectors cannot be assigned to arrays or vice-versa.

```
array[4] real a;
vector[4] b;
row_vector[4] c;
// ...
a = b; // illegal assignment of vector to array
b = a; // illegal assignment of array to vector
a = c; // illegal assignment of row vector to array
c = a; // illegal assignment of array to row vector
```

Mixing row and column vectors

It is not even legal to assign row vectors to column vectors or vice versa.

```
vector[4] b;
row_vector[4] c;
// ...
b = c; // illegal assignment of row vector to column vector
c = b; // illegal assignment of column vector to row vector
```

Mixing matrices and arrays

The same holds for matrices, where 2-dimensional arrays may not be assigned to matrices or vice-versa.

```
array[3, 4] real a;
matrix[3, 4] b;
// ...
a = b;  // illegal assignment of matrix to array
b = a;  // illegal assignment of array to matrix
```

Mixing matrices and vectors

A $1 \times N$ matrix cannot be assigned a row vector or vice versa.

```
matrix[1, 4] a;
row_vector[4] b;
// ...
a = b;  // illegal assignment of row vector to matrix
b = a;  // illegal assignment of matrix to row vector
```

Similarly, an $M \times 1$ matrix may not be assigned to a column vector.

```
matrix[4, 1] a;
vector[4] b;
// ...
a = b;  // illegal assignment of column vector to matrix
b = a;  // illegal assignment of matrix to column vector
```

Size declaration restrictions

An integer expression is used to pick out the sizes of arrays. The same restrictions as for vector and matrix sizes apply, namely that the size is declared with an integer-denoting expression that does not contain any parameters, transformed parameters, or generated quantities.

Size zero arrays

If any of an array's dimensions is size zero, the entire array will be of size zero. That is, if we declare

```
array[3, 0] real a;
```

then the resulting size of a is zero and querying any of its dimensions at run time will result in the value zero. Declared as above, `a[1]` will be a size-zero one-dimensional array. For comparison, declaring

```
array[0, 3] real b;
```

also produces an array with an overall size of zero, but in this case, there is no way to index legally into `b`, because `b[0]` is undefined. The array will behave at run time as if it's a 0×0 array. For example, the result of `to_matrix(b)` will be a 0×0 matrix, not a 0×3 matrix.

5.7. Tuple data type

Stan supports tuples of arbitrary size. The values in a tuple can be of arbitrary type, but the component types must be declared along with the declaration of the tuple. Tuples can be manipulated as a whole, or their elements may be accessed and set

individually.

Declaring tuple variables

Tuples are declared with the keyword `tuple` followed by a parenthesized sequence of types, which determine the types of the respective tuple entries. For example, a tuple with three elements may be declared as

```
tuple(int, vector[3], complex) abc;
```

Tuples must have at least two entries, so the following declarations are illegal.

```
tuple() nil;  // ILLEGAL
tuple(int) n;  // ILLEGAL
```

Tuples can be assigned as a whole if their elements can be assigned individually. For example, `a` can be assigned to `b` in the following example because `int` can be promoted to `complex`.

```
tuple(int, real) a;
...
tuple(complex, real) b = a;
```

Tuple types may have elements which are declared as tuples, such as the following example.

```
tuple(int, tuple(real, complex)) x;
```

In this case, it would probably be simpler to use a 3-tuple type, `tuple(int, real, complex)`.

Tuples can be declared with constraints anywhere that ordinary variables can (i.e., as top-level block variables). That means any context in which it is legal to have a declaration

```
real<lower=0> sigma;
real<lower=0, upper=1> theta;
```

it is legal to have a tuple with constraints such as

```
tuple(real<lower=0>, real<lower=0, upper=1>) sigma_theta;
```

Accessing tuple elements

Tuple elements may be accessed directly. For example, with our declaration of `abc` from the last section, Stan uses `abc.1` for the first element, `abc.2` for the second, and `abc.3` for the third. These numbers must be integer literals (i.e., they cannot

be variables), and must be within the size of the number of elements of tuples. The types of elements are as declared, so that `abc.1` is of type `int`, `abc.2` of type `vector[3]` and `abc.3` of type `complex`.

Assigning tuple elements

Tuple elements can be assigned individually, allowing, e.g.,

```
tuple(int, real) ab;
ab.1 = 123;
ab.2 = 12.9;
```

As with other assignments, promotions will happen if necessary (of `int` to `real` and of `real` to `complex`, along with the corresponding container type promotions).

Unpacking assignment of tuples

For convenience of using values stored in tuples, Stan supports "unpacking" (or "destructuring") of tuples in an assignment statement.

Given a tuple `t` of type `tuple(T1, ..., Tn)` and a sequence of assignable expressions of types `v1`, ..., `vn`, where each `vi` has a type which is assignable from type `Ti`, individual elements of the tuple may be assigned to the corresponding variables in the sequence by the statement

```
(v1, /*...*/, vn) = t;
```

Note that the above parenthesis are required, unlike in some other languages with similar features (e.g., Python).

These unpacking assignments can be nested if the tuple on the right hand side contains nested tuples.

For example, if `T` is a tuple of type `tuple(int, (real, real), complex)`, then the program

```
int i;
real x, y;
complex z;

(i, (x, y), z) = T;
```

Assigns the result of `T.1` to `i`, the result of `T.2.1` to `x`, the result of `T.2.2` to `y`, and the result of `T.3` to `z`.

The left hand side must match in size the tuple on the right. Additionally, the same

variable may not appear more than once in the left hand side of an unpacking assignment.

5.8. Variable types vs. constraints and sizes

The type information associated with a variable only contains the underlying type and dimensionality of the variable.

Type information excludes sizes

The size associated with a given variable is not part of its data type. For example, declaring a variable using

```
array[3] real a;
```

declares the variable a to be an array. The fact that it was declared to have size 3 is part of its declaration, but not part of its underlying type.

When are sizes checked?

Sizes are determined dynamically (at run time) and thus cannot be type-checked statically when the program is compiled. As a result, any conformance error on size will raise a run-time error. For example, trying to assign an array of size 5 to an array of size 6 will cause a run-time error. Similarly, multiplying an $N \times M$ by a $J \times K$ matrix will raise a run-time error if $M \neq J$.

Type information excludes constraints

Like sizes, constraints are not treated as part of a variable's type in Stan when it comes to the compile-time check of operations it may participate in. Anywhere Stan accepts a matrix as an argument, it will syntactically accept a correlation matrix or covariance matrix or Cholesky factor. Thus a covariance matrix may be assigned to a matrix and vice-versa.

Similarly, a bounded real may be assigned to an unconstrained real and vice-versa.

When are function argument constraints checked?

For arguments to functions, constraints are sometimes, but not always checked when the function is called. Exclusions include C++ standard library functions. All probability functions and cumulative distribution functions check that their arguments are appropriate at run time as the function is called.

When are declared variable constraints checked?

For data variables, constraints are checked after the variable is read from a data file or other source. For transformed data variables, the check is done after the statements in the transformed data block have executed. Thus it is legal for intermediate

values of variables to not satisfy declared constraints.

For parameters, constraints are enforced by the transform applied and do not need to be checked. For transformed parameters, the check is done after the statements in the transformed parameter block have executed.

For all blocks defining variables (transformed data, transformed parameters, generated quantities), real values are initialized to `NaN` and integer values are initialized to the smallest legal integer (i.e., a large absolute value negative number).

For generated quantities, constraints are enforced after the statements in the generated quantities block have executed.

Type naming notation

In order to refer to data types, it is convenient to have a way to refer to them. The type naming notation outlined in this section is not part of the Stan programming language, but rather a convention adopted in this document to enable a concise description of a type.

Because size information is not part of a data type, data types will be written without size information. For instance, `array[] 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 `array[,,] int`, indicating the number slots available for indexing. Similarly, `array[,] vector` is the type of a two-dimensional array of vectors.

5.9. Variable declaration

Variables in Stan are declared by giving a type and a name. For example

```
int N;
vector[N] y;
array[5] matrix[3, 4] A;
```

declares a variable `N` that is an integer, a variable `y` that is a vector of length `N` (the previously declared variable), and a variable `A`, which is a length-5 array where each element is a 3 by 4 matrix.

The size of top-level variables in the `parameters`, `transformed parameters`, and `generated quantities` must remain constant across all iterations, therefore only data variables can be used in top-level size declarations.

```
// illegal and will be flagged by the compiler:
generated quantities {
  int N = 10;
```

```
  array[N] int foo;
```

Depending on where the variable is declared in the Stan program, it either must or cannot have size information, and constraints are either optional or not allowed.

```
// valid block variables, but not locals or function parameters
vector<lower=0>[N] u;

// valid as a block or local variable, but not a function parameter
array[3] int is;

// function parameters exclude sizes and cannot be constrained
void pretty_print_tri_lower(matrix x) { ... }
```

Top-level variables can have constraints and must include sizes for their types, as in the above examples. Local variables, like those defined inside loops or local blocks cannot be constrained, but still include sizes. Finally, variables declared as function parameters are not constrained types and *exclude* sizes.

In the following table, the leftmost column is a list of the unconstrained and undimensioned basic types; these are used as function return types and argument types. The middle column is of unconstrained types with dimensions; these are used as local variable types. The variables `M` and `N` indicate number of columns and rows, respectively. The variable `K` is used for square matrices, i.e., `K` denotes both the number of rows and columns. The rightmost column lists the corresponding constrained types. An expression of any right-hand column type may be assigned to its corresponding left-hand column basic type. At runtime, dimensions are checked for consistency for all variables; containers of any sizes may be assigned to function arguments. The constrained matrix types `cov_matrix[K]`, `corr_matrix[K]`, `cholesky_factor_cov[K]`, and `cholesky_factor_corr[K]` are only assignable to matrices of dimensions `matrix[K, K]` types.

Function Argument (unsized)	`Local` (unconstrained)	`Block` (constrained)

`int`	`int`	`int`
		`int<lower=L>`
		`int<upper=U>`
		`int<lower=L, upper=U>`
		`int<offset=O>`

Function Argument (unsized)	Local (unconstrained)	Block (constrained)
		int<multiplier=M>
		int<offset=O, multiplier=M>
real	real	real
		real<lower=L>
		real<upper=U>
		real<lower=L, upper=U>
		real<offset=O>
		real<multiplier=M>
		real<offset=O, multiplier=M>
complex	complex	complex
vector	vector[N]	vector[N]
		vector[N]<lower=L>
		vector[N]<upper=U>
		vector[N]<lower=L, upper=U>
		vector[N]<offset=O>
		vector[N]<multiplier=M>
		vector[N]<offset=O, multiplier=M>
		ordered[N]
		positive_ordered[N]
		simplex[N]
		unit_vector[N]
row_vector	row_vector[N]	row_vector[N]
		row_vector[N]<lower=L>
		row_vector[N]<upper=U>
		row_vector[N]<lower=L, upper=U>
		row_vector[N]<offset=O>
		row_vector[N]<multiplier=M>
		row_vector[N]<offset=O, multiplier=M>
matrix	matrix[M, N]	matrix[M, N]
		matrix[M, N]<lower=L>
		matrix[M, N]<upper=U>
		matrix[M, N]<lower=L, uppers=U>

Function Argument (unsized)	Local (unconstrained)	Block (constrained)
		matrix[M, N]<offset=O>
		matrix[M, N]<multiplier=M>
		matrix[M, N]<offset=O, multiplier=M>
	matrix[K, K]	corr_matrix[K]
	matrix[K, K]	cov_matrix[K]
	matrix[K, K]	cholesky_factor_corr[K]
	matrix[K, K]	cholesky_factor_cov[K]
complex_vector	complex_vector[M]	complex_vector[M]
complex_row_vector	complex_row_vector[N]	complex_row_vector[N]
complex_matrix	complex_matrix[M, N]	complex_matrix[M,N]
array[] vector	array[M] vector[N]	array[M] vector[N]
		array[M] vector[N]<lower=L>
		array[M] vector[N]<upper=U>
		array[M] vector[N]<lower=L, upper=U>
		array[M] vector[N]<offset=O>
		array[M] vector[N]<multiplier=M>
		array[M] vector[N]<offset=O, multiplier=M>
		array[M] ordered[N]
		array[M] positive_ordered[N]
		array[M] simplex[N]
		array[M] unit_vector[N]

Additional array types follow the same basic template as the final example in the table and can contain any of the previous types. The unsized version of arrays with more than one dimension is specified by using commas, e.g. `array[ , ]` is a 2-D array.

For more on how function arguments and return types are declared, consult the User's Guide chapter on functions.

5.10. Compound variable declaration and definition

Stan allows assignable variables to be declared and defined in a single statement. Assignable variables are

- local variables, and
- variables declared in the transformed data, transformed parameters, or generated quantities blocks.

For example, the statement

```
int N = 5;
```

declares the variable `N` to be an integer scalar type and at the same time defines it to be the value of the expression `5`.

Assignment typing

The type of the expression on the right-hand side of the assignment must be assignable to the type of the variable being declared. For example, it is legal to have

```
real sum = 0;
```

even though `0` is of type `int` and `sum` is of type `real`, because integer-typed scalar expressions can be assigned to real-valued scalar variables. In all other cases, the type of the expression on the right-hand side of the assignment must be identical to the type of the variable being declared.

Variables of any type may have values assigned to them. For example,

```
matrix[3, 2] a = b;
```

declares a 3×2 matrix variable `a` and assigns a copy of the value of `b` to the variable `a`. The variable `b` must be of type `matrix` for the statement to be well formed. For the code to execute successfully, `b` must be the same shape as `a`, but this cannot be validated until run time. Because a copy is assigned, subsequent changes to `a` do not affect `b` and subsequent changes to `b` do not affect `a`.

Right-hand side expressions

The right-hand side may be any expression which has a type which is assignable to the variable being declared. For example,

```
matrix[3, 2] a = 0.5 * (b + c);
```

assigns the matrix variable `a` to half of the sum of `b` and `c`. The only requirement

on `b` and `c` is that the expression `b + c` be of type `matrix`. For example, `b` could be of type `matrix` and `c` of type `real`, because adding a matrix to a scalar produces a matrix, and the multiplying by a scalar produces another matrix.

Similarly,

```
complex z = 2 + 3i;
```

assigns the the complex number $2 + 3i$ to the complex scalar `z`. The right-hand side expression can be a call to a user defined function, allowing general algorithms to be applied that might not be otherwise expressible as simple expressions (e.g., iterative or recursive algorithms).

Scope within expressions

Any variable that is in scope and any function that is available in the block in which the compound declaration and definition appears may be used in the expression on the right-hand side of the compound declaration and definition statement.

5.11. Declaring multiple variables at once

Stan will interpret multiple comma-separated variable names following a single type as declaring multiple new variables. This is available for all variable declarations in all blocks.

Types for multiple declarations

The code:

```
real x, y;
```

is equivalent to

```
real x;
real y;
```

As a result, all declarations on the same line must be of the same type.

Combining with other features

The ability to declare multiple variables can be combined with assignments whenever a declare-define is valid, as documented in the section introducing compound declarations and definitions :

```
real x = 3, y = 5.6;
```

Constrained data types can also be declared together, so long as the constraint for each variable is the same:

```
real<lower=0> x, y;
```

6. Expressions

An expression is the 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), based only on the type of its variables and the types of the functions used in it. If an expressions type cannot be determined statically, the Stan compiler will report the location of the problem.

This chapter covers the syntax, typing, and usage of the various forms of expressions in Stan.

6.1. 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 the section on numerical data types).

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 and optionally an exponent with optional sign. Examples of well-formed real literals include the following.

```
0.0, 1.0, 3.14, -217.9387, 2.7e3, -2E-5, 1.23e+3.
```

The notation `e` or `E` followed by a positive or negative integer denotes a power of 10 to multiply. For instance, `2.7e3` and `2.7e+3` denote 2.7×10^3, whereas `-2E-5` denotes -2×10^{-5}.

Imaginary literals

A number followed by the character `i` denotes an imaginary number and is assigned to the numeric type `complex`. The number preceding `i` may be either a real or integer literal and determines the magnitude of the imaginary number. Examples of well-formed imaginary literals include the following.

```
1i, 2i, -325.786i, 1e10i, 2.87e-10i.
```

Note that the character `i` by itself is *not* a well-formed imaginary literal. The unit imaginary number must be written as `1i`.

Complex literals

Stan does not include complex literals directly, but a real or integer literal can be added to an imaginary literal to derive an expression that behaves like a complex literal. Examples include the following.

```
1 + 2i, -3.2e9 + 1e10i
```

These will be assigned the type `complex`, which is the result of adding a real or integer and a complex number. They will also function like literals in the sense that the C++ compiler is able to reduce them to a single complex constant at compile time.

6.2. Variables

A variable by itself is a well-formed expression of the same type as the variable. Variables in Stan consist of ASCII strings containing only the basic lower-case and upper-case Roman letters, digits, and the underscore (_) character. Variables must start with a letter (`a--z` and `A--Z`) and may not end with two underscores (__).

Examples of legal variable identifiers are as follows.

```
a, a3, a_3, Sigma, my_cpp_style_variable, myCamelCaseVariable
```

Unlike in R and BUGS, variable identifiers in Stan may not contain a period character.

Reserved names

Stan reserves many strings for internal use and these may not be used as the name of a variable. An attempt to name a variable after an internal string results in the `stanc` translator halting with an error message indicating which reserved name was used and its location in the model code.

Model name

The name of the model cannot be used as a variable within the model. This is usually not a problem because the default in `bin/stanc` is to append `_model` to the name of the file containing the model specification. For example, if the model is in file `foo.stan`, it would not be legal to have a variable named `foo_model` when using the default model name through `bin/stanc`. With user-specified model names, variables cannot match the model.

Reserved words from Stan language

The following list contains reserved words for Stan's programming language. Not all of these features are implemented in Stan yet, but the tokens are reserved for future use.

```
for, in, while, repeat, until, if, then, else,
true, false, target, struct, typedef, export,
auto, extern, var, static, lower, upper, offset,
multiplier
```

Variables should not be named after types, either, and thus may not be any of the following.

```
int, real, complex, vector, simplex, unit_vector,
ordered, positive_ordered, row_vector, matrix,
cholesky_factor_corr, cholesky_factor_cov,
corr_matrix, cov_matrix, array
```

The following built in functions are also reserved and cannot be used as variable names:

```
print, reject, profile, fatal_error, target
```

The following block identifiers are reserved and cannot be used as variable names:

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

Reserved distribution names

Variable names will also conflict with the names of distributions suffixed with `_lpdf`, `_lpmf`, `_lcdf`, and `_lccdf`, `_cdf`, and `_ccdf`, such as `normal_lcdf_log`. No user-defined variable can take a name ending in `_lupdf` or `_lupmf` even if a corresponding `_lpdf` or `_lpmf` is not defined.

Using any of these variable names causes the `stanc` translator to halt and report the name and location of the variable causing the conflict.

Reserved names backend languages

Stan primarily generates code in C++, which features its own reserved words. It is legal to name a variable any of the following names, however doing so will lead to it being renamed `_stan_NAME` (e.g. `_stan_public`) behind the scenes (in the generated C++ code).

```
alignas, alignof, and, and_eq, asm, bitand, bitor, bool,
case, catch, char, char16_t, char32_t, class, compl, const,
constexpr, const_cast, decltype, default, delete, do,
double, dynamic_cast, enum, explicit, float, friend, goto,
inline, long, mutable, namespace, new, noexcept, not, not_eq,
nullptr, operator, or, or_eq, private, protected, public,
register, reinterpret_cast, short, signed, sizeof,
static_assert, static_cast, switch, template, this, thread_local,
throw, try, typeid, typename, union, unsigned, using, virtual,
volatile, wchar_t, xor, xor_eq, fvar, STAN_MAJOR, STAN_MINOR,
STAN_PATCH, STAN_MATH_MAJOR, STAN_MATH_MINOR, STAN_MATH_PATCH
```

Legal characters

The legal characters for variable identifiers are given in the following table.

Identifier Characters Table *The alphanumeric characters and underscore in base ASCII are the only legal characters in Stan identifiers.*

characters	ASCII 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. Sticking to this range of ASCII makes Stan compatible with Latin-1 or UTF-8 encodings of these characters, which are byte-for-byte identical to ASCII.

Comments allow ASCII-compatible encoding

Within comments, Stan can work with any ASCII-compatible character encoding, such as ASCII itself, UTF-8, or Latin1. It is up to user shells and editors to display them properly.

6.3. Container expressions

Expressions for the Stan container objects, namely arrays, vectors, row vectors, matrices, and tuples, can all be constructed using expressions.

Vector expressions

Square brackets may be wrapped around a sequence of comma separated primitive expressions to produce a row vector expression. For example, the expression `[ 1, 10, 100 ]` denotes a row vector of three elements with real values 1.0, 10.0, and 100.0. Applying the transpose operator to a row vector expression produces a vector expression. This syntax provides a way declare and define small vectors a single line, as follows.

```
row_vector[2] rv2 =  [ 1, 2 ];
vector[3] v3 = [ 3, 4, 5 ]';
```

The vector expression values may be compound expressions or variable names, so it is legal to write `[ 2 * 3, 1 + 4]` or `[ x, y ]`, providing that `x` and `y` are primitive variables.

Matrix expressions

A matrix expression consists of square brackets wrapped around a sequence of comma separated row vector expressions. This syntax provides a way declare and define a matrix in a single line, as follows.

```
matrix[3, 2] m1 = [ [ 1, 2 ], [ 3, 4 ], [5, 6 ] ];
```

Any expression denoting a row vector can be used in a matrix expression. For example, the following code is valid:

```
vector[2] vX = [ 1, 10 ]';
row_vector[2] vY = [ 100, 1000 ];
matrix[3, 2] m2 = [ vX', vY, [ 1, 2 ]  ];
```

Complex vector and matrix expressions

Complex vector expressions work the same way as real vector expressions. For example, the following are all legal Stan expressions and assignments.

```
complex_vector[3] = [1 + 2i, 3 - 1.7i, 0]';
complex_row_vector[2] = [12, -2i];
complex_matrix[2, 3] = [[1 + 2i, 3 - 1.7i, 0],
                        [3.9 - 1.234i, 176i, 1 + 1i]];
```

No empty vector or matrix expressions

The empty expression `[ ]` is ambiguous and therefore is not allowed and similarly expressions such as `[ [ ] ]` or `[ [ ], [ ] ]` are not allowed.

Array expressions

Curly braces may be wrapped around a sequence of expressions to produce an array expression. For example, the expression `{ 1, 10, 100 }` denotes an integer array of three elements with values 1, 10, and 100. This syntax is particularly convenient to define small arrays in a single line, as follows.

```
array[3] int a = { 1, 10, 100 };
```

The values may be compound expressions, so it is legal to write `{ 2 * 3, 1 + 4 }`. It is also possible to write two dimensional arrays directly, as in the following example.

```
array[2, 3] int b = { { 1, 2, 3 }, { 4, 5, 6 } };
```

This way, `b[1]` is `{ 1, 2, 3 }` and `b[2]` is `{ 4, 5, 6 }`.

Whitespace is always interchangeable in Stan, so the above can be laid out as follows to more clearly indicate the row and column structure of the resulting two dimensional array.

```
array[2, 3] int b = { { 1, 2, 3 },
                      { 4, 5, 6 } };
```

Array expression types

Any type of expression may be used within braces to form an array expression. In the simplest case, all of the elements will be of the same type and the result will be an array of elements of that type. For example, the elements of the array can be vectors, in which case the result is an array of vectors.

```
vector[3] b;
vector[3] c;
// ...
array[2] vector[3] d = { b, c };
```

The elements may also be a mixture of `int` and `real` typed expressions, in which case the result is an array of real values.

```
array[2] real b = { 1, 1.9 };
```

Tuple expressions and types

Stan uses parentheses around a comma-separated sequence of expressions to construct a tuple. For example, we can construct a 2-tuple as follows.

```
tuple(int, vector[3]) xy = (42, [1, 2.9, -1.3]');
```

The expression `42` is of type `int` and the expression `[1, 2.9, -1.3]` is of type `row_vector` so that `[1, 2.9, -1.3]'` is of type `vector` and of size 3. The whole tuple expression `(42, [1, 2.9, -1.3]')` thus has a sized type of `tuple(int, vector[3])` and an unsized type (e.g., for a function argument) of `tuple(int, vector)`.

Stan does *not* support the Python notation with trailing commas, such as `(1, 2, 3, )` for a 3-tuple.

Restrictions on values

There are some restrictions on how array expressions may be used that arise from their types being calculated bottom up and the basic data type and assignment rules of Stan.

Rectangular array expressions only

Although it is tempting to try to define a ragged array expression, all Stan data types are rectangular (or boxes or other higher-dimensional generalizations). Thus the following nested array expression will cause an error when it tries to create a non-rectangular array.

```
{ { 1, 2, 3 }, { 4, 5 } }  // compile time error: size mismatch
```

This may appear to be OK, because it is creating a two-dimensional integer array (`array[,] int`) out of two one-dimensional array integer arrays (`array[] int`). But it is not allowed because the two one-dimensional arrays are not the same size. If the elements are array expressions, this can be diagnosed at compile time. If one or both expressions is a variable, then that won't be caught until runtime.

```
{ { 1, 2, 3 }, m }  // runtime error if m not size 3
```

No empty array expressions

Because there is no way to infer the type of the result, the empty array expression (`{ }`) is not allowed. This does not sacrifice expressive power, because a declaration is sufficient to initialize a zero-element array.

```
array[0] int a;   // a is fully defined as zero element array
```

No zero-tuples or one-tuples

There is no way to declare or construct a zero-tuple or one-tuple in Stan. Tuples must be at least two elements long. The expression `()` does not pick out a zero-tuple—it is ill formed. Similarly, the expression `(1)` is of type `int` rather than a tuple.

6.4. 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.

6.5. Arithmetic and matrix operations on expressions

For integer and real-valued expressions, Stan supports the basic binary arithmetic operations of addition (`+`), subtraction (`-`), multiplication (`*`) and division (`/`) in the usual ways.

For integer expressions, Stan supports the modulus (`%`) binary arithmetic operation. 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
m % 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 the list of functions.

For example, if `y` and `mu` are variables of type `vector` and `Sigma` is a variable of type `matrix`, then `(y - mu)' * Sigma * (y - mu)` is a well-formed expression of type `real`. The type of the complete expression is inferred working outward from the subexpressions. The subexpression(s) `y - mu` are of type `vector` because the variables `y` and `mu` are of type `vector`. The transpose of this expression, the subexpression `(y - mu)'` is of type `row_vector`. Multiplication is left associative and transpose has higher precedence than multiplication, so the above expression is equivalent to the following fully specified form `(((y - mu)') * Sigma) * (y - mu)`.

The type of subexpression `(y - mu)' * Sigma` is inferred to be `row_vector`, being the result of multiplying a row vector by a matrix. The whole expression's type is thus the type of a row vector multiplied by a (column) vector, which produces a `real` value.

Stan provides elementwise matrix multiplication (e.g., `a .* b`) and division (e.g., `a ./ b`) operations. These provide a shorthand to replace loops, but are not intrinsically more efficient than a version programmed with an elementwise calculations and assignments in a loop. For example, given declarations,

```
vector[N] a;
vector[N] b;
vector[N] c;
```

the assignment,

```
c = a .* b;
```

produces the same result with roughly the same efficiency as the loop

```
for (n in 1:N) {
  c[n] = a[n] * b[n];
}
```

Stan supports exponentiation (`^`) of integer and real-valued expressions. The return type of exponentiation is always a real-value. For example, assuming `n` and `m` are integer variables and `x` and `y` real variables, the following expressions are legal.

```
3 ^ 2
3.0 ^ -2
3.0 ^ 0.14
x ^ n
```

```
n ^ x
n ^ m
x ^ y
```

Exponentiation is right associative, so the expression `2 ^ 3 ^ 4` is equivalent to the fully specified form `2 ^ (3 ^ 4)`.

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 the following table.

Operator Precedence Table *Stan's unary, binary, and ternary operators, with their precedences, associativities, place in an expression, and a description. The last two lines list the precedence of function application and array, matrix, and vector indexing. The operators are listed in order of precedence, from least tightly binding to most tightly binding. The full set of legal arguments and corresponding result types are provided in the function documentation for the operators (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.*

Op.	Prec.	Assoc.	Placement	Description
`? ~ :`	10	right	ternary infix	conditional
`||`	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	elementwise multiplication
`/`	4	left	binary infix	(right) division
`./`	4	left	binary infix	elementwise division
`%`	4	left	binary infix	modulus

Op.	Prec.	Assoc.	Placement	Description
`\`	3	left	binary infix	left division
`%/%`	3	left	binary infix	integer division
`!`	2	n/a	unary prefix	logical negation
`-`	2	n/a	unary prefix	negation
`+`	2	n/a	unary prefix	promotion (no-op in Stan)
`^`	1	right	binary infix	exponentiation
`.^`	1	right	binary infix	elementwise exponentiation
`'`	0	n/a	unary postfix	transposition
`()`	0	n/a	prefix, wrap	function application
`[]`	0	left	prefix, wrap	array, matrix indexing

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 and exponentiation bind more tightly than any other arithmetic or logical operation. For vectors, row vectors, and matrices, `-u'` is interpreted as `-(u')`, `u * v'` as `u* (v')`, and `u' * v` as `(u') * v`. For integer and reals, `-n ^ 3` is interpreted as `-(n ^ 3)`.

6.6. Conditional operator

Conditional operator syntax

The ternary conditional operator is unique in that it takes three arguments and uses a mixed syntax. If `a` is an expression of type `int` and `b` and `c` are expressions that can be converted to one another (e.g., compared with `==`), then

```
a ? b : c
```

is an expression of the promoted type of `b` and `c`. The only promotion allowed in Stan is integer -> real -> complex; e.g. if one argument is of type `int` and the other of type `real`, the conditional expression as a whole is of type `real`. In other cases,

the arguments have to be of the same underlying Stan type (i.e., constraints don't count, only the shape) and the conditional expression is of that type.

Conditional operator precedence
The conditional operator is the most loosely binding operator, so its arguments rarely require parentheses for disambiguation. For example,

```
a > 0 || b < 0 ? c + d : e - f
```

is equivalent to the explicitly grouped version

```
(a > 0 || b < 0) ? (c + d) : (e - f)
```

The latter is easier to read even if the parentheses are not strictly necessary.

Conditional operator associativity
The conditional operator is right associative, so that

```
a ? b : c ? d : e
```

parses as if explicitly grouped as

```
a ? b : (c ? d : e)
```

Again, the explicitly grouped version is easier to read.

Conditional operator semantics
Stan's conditional operator works very much like its C++ analogue. The first argument must be an expression denoting an integer. Typically this is a variable or a relation operator, as in the variable `a` in the example above. Then there are two resulting arguments, the first being the result returned if the condition evaluates to true (i.e., non-zero) and the second if the condition evaluates to false (i.e., zero). In the example above, the value `b` is returned if the condition evaluates to a non-zero value and `c` is returned if the condition evaluates to zero.

Lazy evaluation of results
The key property of the conditional operator that makes it so useful in high-performance computing is that it only evaluates the returned subexpression, not the alternative expression. In other words, it is not like a typical function that evaluates its argument expressions eagerly in order to pass their values to the function. As usual, the saving is mostly in the derivatives that do not get computed rather than the unnecessary function evaluation itself.

Promotion to parameter

If one return expression is a data value (an expression involving only constants and variables defined in the data or transformed data block), and the other is not, then the ternary operator will promote the data value to a parameter value. This can cause needless work calculating derivatives in some cases and be less efficient than a full `if-then` conditional statement. For example,

```
data {
  array[10] real x;
  // ...
}
parameters {
  array[10] real z;
  // ...
}
model {
  y ~ normal(cond ? x : z, sigma);
  // ...
}
```

would be more efficiently (if not more transparently) coded as

```
if (cond) {
  y ~ normal(x, sigma);
} else {
  y ~ normal(z, sigma);
}
```

The conditional statement, like the conditional operator, only evaluates one of the result statements. In this case, the variable `x` will not be promoted to a parameter and thus not cause any needless work to be carried out when propagating the chain rule during derivative calculations.

6.7. Indexing

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 `array [] 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 `array[,] 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 `array[„] real`, then `x[2]` is of type `array[,] real`, and `x[2, 3]` is of type `array[] real`. As a result, the expressions `x[2, 3]` and `x[2][3]` have the same meaning.

Accessing matrix rows

If `Sigma` is a variable of type `matrix`, then `Sigma[1]` denotes the first row of `Sigma` and has the type `row_vector`.

Mixing array and vector/matrix indexes

Stan supports mixed indexing of arrays and their vector, row vector or matrix values. For example, if `m` is of type `matrix[ , ]`, a two-dimensional array of matrices, then `m[1]` refers to the first row of the array, which is a one-dimensional array of matrices. More than one index may be used, so that `m[1, 2]` is of type `matrix` and denotes the matrix in the first row and second column of the array. Continuing to add indices, `m[1, 2, 3]` is of type `row_vector` and denotes the third row of the matrix denoted by `m[1, 2]`. Finally, `m[1, 2, 3, 4]` is of type `real` and denotes the value in the third row and fourth column of the matrix that is found at the first row and second column of the array `m`.

6.8. Multiple indexing and range indexing

In addition to single integer indexes, as described in the language indexing section, Stan supports multiple indexing. Multiple indexes can be integer arrays of indexes, lower bounds, upper bounds, lower and upper bounds, or simply shorthand for all of the indexes. A complete table of index types is given in the following table.

Indexing Options Table *Types of indexes and examples with one-dimensional containers of size `N` and an integer array `ii` of type `array [] real` size `K`.*

index type	example	value
integer	`a[11]`	value of `a` at index 11
integer array	`a[ii]`	`a[ii[1]],...,a[ii[K]]`
lower bound	`a[3:]`	`a[3],...,a[N]`
upper bound	`a[:5]`	`a[1],...,a[5]`
range	`a[2:7]`	`a[2],...,a[7]`
all	`a[:]`	`a[1],...,a[N]`

index type	example	value
all	`a[]`	`a[1],...,a[N]`

Multiple index semantics

The fundamental semantic rule for dealing with multiple indexes is the following. If `idxs` is a multiple index, then it produces an indexable position in the result. To evaluate that index position in the result, the index is first passed to the multiple index, and the resulting index used.

```
a[idxs, ...][i, ...] = a[idxs[i], ...][...]
```

On the other hand, if `idx` is a single index, it reduces the dimensionality of the output, so that

```
a[idx, ...] = a[idx][...]
```

The only issue is what happens with matrices and vectors. Vectors work just like arrays. Matrices with multiple row indexes and multiple column indexes produce matrices. Matrices with multiple row indexes and a single column index become (column) vectors. Matrices with a single row index and multiple column indexes become row vectors. The types are summarized in the following table.

Matrix Indexing Table *Special rules for reducing matrices based on whether the argument is a single or multiple index. Examples are for a matrix* `a`*, with integer single indexes* `i` *and* `j` *and integer array multiple indexes* `is` *and* `js`*. The same typing rules apply for all multiple indexes.*

example	row index	column index	result type
`a[i]`	single	n/a	row vector
`a[is]`	multiple	n/a	matrix
`a[i, j]`	single	single	real
`a[i, js]`	single	multiple	row vector
`a[is, j]`	multiple	single	vector
`a[is, js]`	multiple	multiple	matrix

Evaluation of matrices with multiple indexes is defined to respect the following distributivity conditions.

```
m[idxs1, idxs2][i, j] = m[idxs1[i], idxs2[j]]
m[idxs, idx][j] = m[idxs[j], idx]
m[idx, idxs][j] = m[idx, idxs[j]]
```

Evaluation of arrays of matrices and arrays of vectors or row vectors is defined recursively, beginning with the array dimensions.

6.9. Function application

Stan provides a range of built in mathematical and statistical functions, which are documented in the built-in function documentation.

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 `lmultiply` function is

```
real lmultiply(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(array [] 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 the Stan Functions Reference built-in constants section for a list of built-in constants.

Type promotion and function resolution

Because of integer to real type promotion, rules must be established for which function is called given a sequence of argument types. The scheme employed by Stan is the same as that used by C++, which resolves a function call to the function requiring the minimum number of type promotions.

For example, consider a situation in which the following two function signatures have been registered for `foo`.

```
real foo(real, real);
int foo(int, int);
```

The use of `foo` in the expression `foo(1.0, 1.0)` resolves to `foo(real, real)`, and thus the expression `foo(1.0, 1.0)` itself is assigned a type of `real`.

Because integers may be promoted to real values, the expression `foo(1, 1)` could potentially match either `foo(real, real)` or `foo(int, int)`. The former requires two type promotions and the latter requires none, so `foo(1, 1)` is resolved to function `foo(int, int)` and is thus assigned the type `int`.

The expression `foo(1, 1.0)` has argument types `(int, real)` and thus does not explicitly match either function signature. By promoting the integer expression `1` to type `real`, it is able to match `foo(real, real)`, and hence the type of the function expression `foo(1, 1.0)` is `real`.

In some cases (though not for any built-in Stan functions), a situation may arise in which the function referred to by an expression remains ambiguous. For example, consider a situation in which there are exactly two functions named `bar` with the following signatures.

```
real bar(real, int);
real bar(int, real);
```

With these signatures, the expression `bar(1.0, 1)` and `bar(1, 1.0)` resolve to the first and second of the above functions, respectively. The expression `bar(1.0, 1.0)` is illegal because real values may not be demoted to integers. The expression `bar(1, 1)` is illegal for a different reason. If the first argument is promoted to a real

value, it matches the first signature, whereas if the second argument is promoted to a real value, it matches the second signature. The problem is that these both require one promotion, so the function name `bar` is ambiguous. If there is not a unique function requiring fewer promotions than all others, as with `bar(1, 1)` given the two declarations above, the Stan compiler will flag the expression as illegal.

Random-number generating functions

For most of the distributions supported by Stan, there is a corresponding random-number generating function. These random number generators are named by the distribution with the suffix `_rng`. For example, a univariate normal random number can be generated by `normal_rng(0, 1);` only the parameters of the distribution, here a location (0) and scale (1) are specified because the variate is generated.

Random-number generators locations

The use of random-number generating functions is restricted to the transformed data and generated quantities blocks; attempts to use them elsewhere will result in a parsing error with a diagnostic message. They may also be used in the bodies of user-defined functions whose names end in `_rng`.

This allows the random number generating functions to be used for simulation in general, and for Bayesian posterior predictive checking in particular.

Posterior predictive checking

Posterior predictive checks typically use the parameters of the model to generate simulated data (at the individual and optionally at the group level for hierarchical models), which can then be compared informally using plots and formally by means of test statistics, to the actual data in order to assess the suitability of the model; see Chapter 6 of (Gelman et al. 2013) for more information on posterior predictive checks.

6.10. 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, complex, vector, row_vector,  matrix, complex_vector,
complex_row_vector, complex_matrix
```

Every basic declared type corresponds to a primitive type; the following table shows the mapping from types to their primitive types.

Primitive Type Table *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 plus array dimensionality.*

type	primitive type
`int`	`int`
`real`	`real`
`vector`	`vector`
`simplex`	`vector`
`unit_vector`	`vector`
`ordered`	`vector`
`positive_ordered`	`vector`
`row_vector`	`row_vector`
`matrix`	`matrix`
`cov_matrix`	`matrix`
`corr_matrix`	`matrix`
`cholesky_factor_cov`	`matrix`
`cholesky_factor_corr`	`matrix`
`complex_vector`	`complex_vector`
`complex_row_vector`	`complex_row_vector`
`complex_matrix`	`complex_matrix`

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, `array [] real` has an array dimensionality of 1, `int` an array dimensionality of 0, and `array [„] 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.

```
array[I, J, K] matrix[M, N] a;
```

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.

Promotion

There are two basic promotion rules,

1. `int` types may be promoted to `real`, and
2. `real` types may be promoted to `complex`.

Plus, promotion is transitive, so that

3. if type `U` can be promoted to type `V` and type `V` can be promoted to type `T`, then `U` can be promoted to `T`.

The first rule means that expressions of type `int` may be used anywhere an expression of type `real` is specified, namely in assignment or function argument passing. An integer is promoted to real by casting it in the underlying C++ code.

The remaining rules have to do with covariant typing rules, which say that a container of type `U` may be promoted to a container of the same shape of type `T` if `U` can be promoted to `T`. For vector and matrix types, this induces three rules,

4. `vector` may be promoted to `complex_vector`,
5. `row_vector` may be promoted to `complex_row_vector`
6. `matrix` may be promoted to `complex_matrix`.

For array types, there's a single rule

7. `array[...] U` may be promoted to `array[...] T` if `U` can be promoted to `T`.

For example, this means `array[,] int` may be used where `array [,] real` or `array [,] complex` is required; as another example, `array[] real` may be used anywhere `array[] complex` is required.

Tuples have the natural extension of the above rules, applied to all sub-types at once

8. A `tuple(U1, ..., UN)` may be promoted to a `tuple(T1, ..., TN)` if every `Un` can be promoted to `Tn` for `n` in `1:N`

Literals

An integer literal expression such as `42` is of type `int`. Real literals such as `42.0` are of type `real`. Imaginary literals such as `-17i` are of type `complex`. the expression `7`

`- 2i` acts like a complex literal, but technically it combines a real literal `7` and an imaginary literal `2i` through subtraction.

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 in each scope 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, i2, ..., iN]` is the same as that of `e[i1][i2]...[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 type `array[i1, i2, ..., iN] T` and `k`, `i1`, ..., `iN` are expressions of type `int`, then `e[k]` is an expression of type `array[i2, ..., iN] T`,
- if `e` is an expression of type `array[i] T` with `i` and `k` expressions of type `int`, then `e[k]` is of type `T`,
- if `e` has implementation type `vector` or `row_vector`, dimensionality 0, then `e[i]` has implementation type `real`,
- if `e` has implementation type `matrix`, then `e[i]` has type `row_vector`,
- if `e` has implementation type `complex_vector` or `complex_row_vector` and `i` is an expression of type `int`, then `e[i]` is an expression of type `complex`, and
- if `e` has implementation type `complex_matrix`, and `i` is an expression of type `int`, then `e[i]` is an expression of type `complex_row_vector`.

Function application
If `f` is the name of a function and `e1`,...,`eN` are expressions for $N \geq 0$, then `f(e1,...,eN)` is an expression whose type is determined by the return type in the function signature for `f` given `e1` through `eN`. Recall that a function signature is a declaration of the argument types and the result type.

In looking up functions, binary operators like `real * real` are defined as `operator*(real, real)` in the documentation and index.

In matching a function definition, all of the promotion rules are in play (integers

[1] Languages such as C++ and R allow the declaration of a variable of a given name in a narrower scope to hide (take precedence over for evaluation) a variable defined in a containing scope.

may be promoted to reals, reals to complex, and containers may be promoted if their types are promoted). For example, arguments of type `int` may be promoted to type `real` or `complex` if necessary (see the subsection on type promotion in the function application section, a `real` argument will be promoted to `complex` if necessary, a `vector` will be promoted to `complex_vector` if necessary, and so on.

In general, matrix operations return the lowest inferable 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)`.

6.11. Higher-order functions

There are several expression constructions in Stan that act as higher-order functions.[2]

The higher-order functions and the signature of their argument functions are listed in the following pair of tables.

Higher-order Functions Table *Higher-order functions in Stan with their argument function types. The first group of arguments can be a function of parameters or data. The second group of arguments, consisting of a real and integer array in all cases, must be expressions involving only data and literals.*

function	parameter or data args	data args	return type
`algebra_solver`	`vector, vector`	`array [] real, array [] real`	`vector`
`algebra_solver_newton`	`vector, vector`	`array [] real, array [] real`	`vector`
`integrate_1d`	`real, real, array [] real`	`array [] real, array [] real`	`real`
`integrate_ode_X`	`real, array [] real, array [] real`	`array [] real, array [] real`	`array [] real`
`map_rect`	`vector, vector`	`array [] real, array [] real`	`vector`

For example, the `integrate_ode_rk45` function can be used to integrate differential equations in Stan:

[2] Internally, they are implemented as their own expression types because Stan doesn't have object-level functional types (yet).

```
functions {
  array [] real foo(real t,
                    array [] real y,
                    array [] real theta,
                    array [] real x_r,
                    array [] real x_i) {
    // ...
  }
}
// ...
int<lower=1> T;
array[2] real y0;
real t0;
array[T] real ts;
array[1] real theta;
array[0] real x_r;
array[0] int x_i;
// ...
array[T, 2] real y_hat = integrate_ode_rk45(foo, y0, t0,
                                            ts, theta, x_r, x_i);
```

The function argument is `foo`, the name of the user-defined function; as shown in the higher-order functions table, `integrate_ode_rk45` takes a real array, a real, three more real arrays, and an integer array as arguments and returns 2D real array.

Variadic Higher-order Functions Table *Variadic Higher-order functions in Stan with their argument function types. The first group of arguments are restricted in type. The sequence of trailing arguments can be of any length with any types.*

function	restricted args	return type
`solve_X`	`vector`	`vector`
`ode_X,`	`vector, real, array [] real`	`vector[]`
`reduce_sum`	`array[] T, T1, T2`	`real`

`T`, `T1`, and `T2` can be any Stan type.

For example, the `ode_rk45` function can be used to integrate differential equations in Stan:

```
functions {
  vector foo(real t, vector y, real theta, vector beta,
             array [] real x_i, int index) {
    // ...
  }
}
// ...
int<lower=1> T;
vector[2] y0;
real t0;
array[T] real ts;
real theta;
vector[7] beta;
array[10] int x_i;
int index;
// ...
vector[2] y_hat[T] = ode_rk45(foo, y0, t0, ts, theta,
                              beta, x_i, index);
```

The function argument is `foo`, the name of the user-defined function. As shown in the variadic higher-order functions table, `ode_rk45` takes a real, a vector, a real, a real array, and a sequence of arguments whose types match those at the end of `foo` and returns an array of vectors.

Functions passed by reference

The function argument to higher-order functions is always passed as the first argument. This function argument must be provided as the name of a user-defined or built-in function. No quotes are necessary.

Data-restricted arguments

Some of the arguments to higher-order functions are restricted to data. This means they must be expressions containing only data variables, transformed data variables, or literals; the may contain arbitrary functions applied to data variables or literals, but must not contain parameters, transformed parameters, or local variables from any block other than transformed data.

For user-defined functions the qualifier `data` may be prepended to the type to restrict the argument to data-only variables.

6.12. Chain rule and derivatives

Derivatives of the log probability function defined by a model are used in several ways by Stan. The Hamiltonian Monte Carlo samplers, including NUTS, use gradients to guide updates. The BFGS optimizers also use gradients to guide search for posterior modes.

Errors due to chain rule

Unlike evaluations in pure mathematics, evaluation of derivatives in Stan is done by applying the chain rule on an expression-by-expression basis, evaluating using floating-point arithmetic. As a result, models such as the following are problematic for inference involving derivatives.

```
parameters {
  real x;
}
model {
  x ~ normal(sqrt(x - x), 1);
}
```

Algebraically, the distribution statement in the model could be reduced to

```
  x ~ normal(0, 1);
```

and it would seem the model should produce unit normal draws for `x`. But rather than canceling, the expression `sqrt(x - x)` causes a problem for derivatives. The cause is the mechanistic evaluation of the chain rule,

$$\begin{aligned}
\frac{d}{dx}\sqrt{x-x} &= \frac{1}{2\sqrt{x-x}} \times \frac{d}{dx}(x-x) \\
&= \frac{1}{0} \times (1-1) \\
&= \infty \times 0 \\
&= \text{NaN}.
\end{aligned}$$

Rather than the $x - x$ canceling out, it introduces a 0 into the numerator and denominator of the chain-rule evaluation.

The only way to avoid this kind problem is to be careful to do the necessary algebraic reductions as part of the model and not introduce expressions like `sqrt(x - x)` for which the chain rule produces not-a-number values.

Diagnosing problems with derivatives

The best way to diagnose whether something is going wrong with the derivatives is to use the test-gradient option to the sampler or optimizer inputs; this option is available in both Stan and RStan (though it may be slow, because it relies on finite differences to make a comparison to the built-in automatic differentiation).

For example, compiling the above model to an executable `sqrt-x-minus-x` in CmdStan, the test can be run as

```
> ./sqrt-x-minus-x diagnose test=gradient
```

which produces

```
...
TEST GRADIENT MODE

 Log probability=-0.393734

 param idx           value           model     finite diff           error
         0       -0.887393             nan               0             nan
```

Even though finite differences calculates the right gradient of 0, automatic differentiation follows the chain rule and produces a not-a-number output.

7. Statements

The blocks of a Stan program are made up of variable declarations and statements; see the blocks chapter for details. 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.

The basis of Stan's execution is the evaluation of a log probability function (specifically, a log probability density function) for a given set of (real-valued) parameters. Log probability functions can be constructed by using distribution statements and log probability increment statements. 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.

7.1. Statement block contexts

The data and parameters blocks do not allow statements of any kind because these blocks are solely used to declare the data variables for input and the parameter variables for sampling. All other blocks allow statements. In these blocks, both variable declarations and statements are allowed. All top-level variables in a block are considered block variables. See the blocks chapter for more information about the block structure of Stan programs.

7.2. Assignment statements

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 as follows.

```
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 to subcomponents of larger multi-variate data structures are supported by Stan. For example, `a` is an array of type `array[,] real` and `b` is an array of type `array[] 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;
```

Promotion

Stan allows assignment of lower types to higher types, but not vice-versa. That is, we can assign an expression of type `int` to an lvalue of type `real`, and we can assign an expression of type `real` to an lvalue of type `complex`. Furthermore, promotion is transitive, so that we can assign an expression of type `int` to an lvalue of type

`complex`.

Promotion extends to containers, so that arrays of `int` can be promoted to arrays of `real` during assignment, and arrays of `real` can be assigned to an lvalue of type array of `complex`. Similarly, an expression of type `vector` may be assigned to an lvalue of type `complex_vector`, and similarly for row vectors and matrices.

Lvalue summary

The expressions that are legal left-hand sides of assignment statements are known as "lvalues." In Stan, there are three kinds of legal lvalues,

- a variable, or
- a variable with one or more indices, or
- a comma separated list of lvalues surrounded by `(` and `)`

To be used as an lvalue, an indexed variable must have at least as many dimensions as the number of indices provided. An array of real or integer types has as many dimensions as it is declared for. A matrix has two dimensions and a vector or row vector one dimension; this also holds for the constrained types, covariance and correlation matrices and their Cholesky factors and ordered, positive ordered, and simplex vectors. An array of matrices has two more dimensions than the array and an array of vectors or row vectors has one more dimension than the array. Note that the number of indices can be less than the number of dimensions of the variable, meaning that the right hand side must itself be multidimensional to match the remaining dimensions.

Multiple indexes

Multiple indexes, as described in the multi-indexing section, are also permitted on the left-hand side of assignments. Indexing on the left side works exactly as it does for expressions, with multiple indexes preserving index positions and single indexes reducing them. The type on the left side must still match the type on the right side.

Aliasing

All assignment is carried out as if the right-hand side is copied before the assignment. This resolves any potential aliasing issues arising from he right-hand side changing in the middle of an assignment statement's execution.

Compound arithmetic and assignment statement

Stan's arithmetic operators may be used in compound arithmetic and assignment operations. For example, consider the following example of compound addition and assignment.

```
real x = 5;
x += 7;  // value of x is now 12
```

The compound arithmetic and assignment statement above is equivalent to the following long form.

```
x = x + 7;
```

In general, the compound form

```
x op= y
```

will be equivalent to

```
x = x op y;
```

The compound statement will be legal whenever the long form is legal. This requires that the operation `x op y` must itself be well formed and that the result of the operation be assignable to `x`. For the expression `x` to be assignable, it must be an indexed variable where the variable is defined in the current block. For example, the following compound addition and assignment statement will increment a single element of a vector by two.

```
vector[N] x;
x[3] += 2;
```

As a further example, consider

```
matrix[M, M] x;
vector[M] y;
real z;
x *= x;  // OK, (x * x) is a matrix
x *= z;  // OK, (x * z) is a matrix
x *= y;  // BAD, (x * y) is a vector
```

The supported compound arithmetic and assignment operations are listed in the compound arithmetic/assignment table; they are also listed in the index prefaced by `operator`, e.g., `operator+=`.

Compound Arithmetic/Assignment Table. *Stan allows compound arithmetic and assignment statements of the forms listed in the table. The compound form is legal whenever the corresponding long form would be legal and it has the same effect.*

operation	compound	unfolded
addition	`x += y`	`x = x + y`
subtraction	`x -= y`	`x = x - y`
multiplication	`x *= y`	`x = x * y`
division	`x /= y`	`x = x / y`
elementwise multiplication	`x .*= y`	`x = x .* y`
elementwise division	`x ./= y`	`x = x ./ y`

7.3. Increment log density

The basis of Stan's execution is the evaluation of a log probability function (specifically, a log probability density function) for a given set of (real-valued) parameters; this function returns the log density of the posterior up to an additive constant. Data and transformed data are fixed before the log density is evaluated. The total log probability is initialized to zero. Next, any log Jacobian adjustments accrued by the variable constraints are added to the log density (the Jacobian adjustment may be skipped for optimization). Distribution statements and log probability increment statements may add to the log density in the model block. A log probability increment statement directly increments the log density with the value of an expression as follows.[1]

```
target += -0.5 * y * y;
```

The keyword `target` here is actually not a variable, and may not be accessed as such (though see below on how to access the value of target through a special function).

In this example, the unnormalized log probability of a unit normal variable y is added to the total log probability. In the general case, the argument can be any expression.[2]

An entire Stan model can be implemented this way. For instance, the following model has a single variable according to a unit normal probability.

[1]The current notation replaces two previous versions. Originally, a variable `lp__` was directly exposed and manipulated; this is no longer allowed. The original statement syntax for `target += u` was `increment_log_prob(u)`, but this form was removed in Stan 2.33

[2]Writing this model with the expression `-0.5 * y * y` is more efficient than with the equivalent expression `y * y / -2` because multiplication is more efficient than division; in both cases, the negation is rolled into the numeric literal (`-0.5` and `-2`). Writing `square(y)` instead of `y * y` would be even more efficient because the derivatives can be precomputed, reducing the memory and number of operations required for automatic differentiation.

```
parameters {
  real y;
}
model {
  target += -0.5 * y * y;
}
```

This model defines a log probability function

$$\log p(y) = -\frac{y^2}{2} - \log Z$$

where Z is a normalizing constant that does not depend on y. The constant Z is conventionally written this way because on the linear scale,

$$p(y) = \frac{1}{Z} \exp\left(-\frac{y^2}{2}\right).$$

which is typically written without reference to Z as

$$p(y) \propto \exp\left(-\frac{y^2}{2}\right).$$

Stan only requires models to be defined up to a constant that does not depend on the parameters. This is convenient because often the normalizing constant Z is either time-consuming to compute or intractable to evaluate.

Built in distributions

The built in distribution functions in Stan are all available in normalized and unnormalized form. The normalized forms include all of the terms in the log density, and the unnormalized forms drop terms which are not directly or indirectly a function of the model parameters.

For instance, the `normal_lpdf` function returns the log density of a normal distribution:

$$\text{normal\_lpdf}(x|\mu,\sigma) = -\log\left(\sigma\sqrt{2\pi}\right) - \frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2$$

The `normal_lupdf` function returns the log density of an unnormalized distribution. With the unnormalized version of the function, Stan does not define what the normalization constant will be, though usually as many terms as possible are dropped

to make the calculation fast. Dropping a constant `sigma` term, `normal_lupdf` would be equivalent to:

$$\text{normal\_lupdf}(x|\mu,\sigma) = -\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2$$

All functions ending in `_lpdf` have a corresponding `_lupdf` version which evaluates and returns the unnormalized density. The same is true for `_lpmf` and `_lupmf`.

Relation to compound addition and assignment
The increment log density statement looks syntactically like compound addition and assignment (see the compound arithmetic/assignment section, it is treated as a primitive statement because `target` is not itself a variable. So, even though

```
target += lp;
```

is a legal statement, the corresponding long form is not legal.

```
target = target + lp;  // BAD, target is not a variable
```

Vectorization
The `target += ...` statement accepts an argument in place of `...` for any expression type, including integers, reals, vectors, row vectors, matrices, and arrays of any dimensionality, including arrays of vectors and matrices. For container arguments, their sum will be added to the total log density.

Accessing the log density
To access accumulated log density up to the current execution point, the function `target()` may be used.

7.4. Sampling statements

The term "sampling statement" has been replaced with distribution statement.

7.5. Distribution statements

Stan supports writing probability statements also using distribution statements, for example

```
y ~ normal(mu, sigma);
mu ~ normal(0, 10);
sigma ~ normal(0, 1);
```

The symbol $\sim$ is called tilde. Due to historical reasons, the distribution statements

used to be called "sampling statements" in Stan, but that term is not recommended anymore as it is a less accurate description.

In general, we can read $\sim$ as "is distributed as," and overall this notation is used as a shorthand for defining distributions, so that the above example can be written also as

$$\begin{aligned} p(y|\mu,\sigma) &= \text{normal}(y|\mu,\sigma) \\ p(\mu) &= \text{normal}(\mu|0,10) \\ p(\sigma) &= \text{normal}^{+}(\sigma|0,1). \end{aligned}$$

A collection of distribution statements define a joint distribution as the product of component distributions

$$p(y,\mu,\sigma) = p(y|\mu,\sigma)p(\mu)p(\sigma).$$

This works even if the model is not constructed generatively. For example, suppose you include the following code in a Stan model:

```
a ~ normal(0, 1);
a ~ normal(0, 1);
```

This is translated to

$$p(a) = \text{normal}(a|0,1)\text{normal}(a|0,1),$$

which in this case is $\text{normal}(a|0,1/\sqrt{2})$. One might expect that the above two lines of code would represent a redundant expression of a $\text{normal}(a|0,1)$ prior, but, no, each line of code corresponds to an additional term in the target, or log posterior density. You can think of each line as representing an additional piece of information.

When the joint distribution is considered as a function of parameters (e.g. μ, σ) given fixed data, it is proportional to the posterior distribution. In general, the posterior distribution is not a normalized probability density function—that is, it will be positive but will not in general integrate to 1—but the proportionality is sufficient for the Stan algorithms.

Stan always constructs the target function—in Bayesian terms, the log posterior density function of the parameter vector—by adding terms in the model block. Equivalently, each $\sim$ statement corresponds to a multiplicative factor in the unnormalized posterior density.

Distribution statements (~) accept only built-in or user-defined distributions on the right side. The left side of a distribution statement may be data, parameter, or

a complex expression, but the evaluated type needs to match one of the allowed types of the distribution on the right (see more below).

In Stan, a distribution statement is merely a notational convenience following the typical notation used to present models in the literature. The above model defined with distribution statements could be expressed as a direct increment on the total log probability density as

```
target += normal_lpdf(y | mu, sigma);
target += normal_lpdf(mu | 0, 10);
target += normal_lpdf(sigma | 0, 1);
```

Stan models can mix distribution statements and log probability increment statements. Although statistical models are usually defined with distributions in the literature, there are several scenarios in which we may want to code the log likelihood or parts of it directly, for example, due to computational efficiency (e.g. censored data model) or coding language limitations (e.g. mixture models in Stan). This is possible with log probability increment statements. See also the discussion below about Jacobians.

In general, a distribution statement of the form

```
y ~ dist(theta1, ..., thetaN);
```

involving subexpressions `y` and `theta1` through `thetaN` (including the case where `N` is zero) will be well formed if and only if the corresponding log probability increment statement is well-formed. For densities allowing real `y` values, the log probability density function is used,

```
target += dist_lpdf(y | theta1, ..., thetaN);
```

For those restricted to integer `y` values, the log probability mass function is used,

```
target += dist_lpmf(y | theta1, ..., thetaN);
```

This will be well formed if and only if `dist_lpdf(y | theta1, ..., thetaN)` or `dist_lpmf(y | theta1, ..., thetaN)` is a well-formed expression of type `real`. User defined distributions can be defined in functions block by using function names ending with `_lpdf`.

Log probability increment vs. distribution statement

Although both lead to the same inference algorithm behavior in Stan, there is one critical difference between using the distribution statement, as in

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

and explicitly incrementing the log probability function, as in

```
target += normal_lpdf(y | mu, sigma);
```

The distribution statement drops all the terms in the log probability function that are constant, whereas the explicit call to `normal_lpdf` adds all of the terms in the definition of the log normal probability function, including all of the constant normalizing terms. Therefore, the explicit increment form can be used to recreate the exact log probability values for the model. Otherwise, the distribution statement form will be faster if any of the input expressions, `y`, `mu`, or `sigma`, involve only constants, data variables, and transformed data variables. See the section Built in distributions above discussing `_lupdf` and `_lupmf` functions that also drops all the constant terms.

User-transformed variables

The left-hand side of a distribution statement may be an arbitrary expression (of compatible type)". For instance, it is legal syntactically to write

```
parameters {
  real<lower=0> beta;
}
// ...
model {
  log(beta) ~ normal(mu, sigma);
}
```

Unfortunately, this is not enough to properly model `beta` as having a lognormal distribution. Whenever a nonlinear transform is applied to a parameter, such as the logarithm function being applied to `beta` here, and then used on the left-hand side of a distribution statement or on the left of a vertical bar in a log pdf function, an adjustment must be made to account for the differential change in scale and ensure `beta` gets the correct distribution. The correction required is to add the log Jacobian of the transform to the target log density; see the change of variables section for full definitions. For the case above, the following adjustment will account for the log transform.[3]

```
target += - log(abs(y));
```

[3] Because $\log \left| \frac{d}{dy} \log y \right| = \log |1/y| = -\log |y|$.

Truncated distributions

Stan supports truncating distributions with lower bounds, upper bounds, or both.

Truncating with lower and upper bounds

A probability density function $p(x)$ for a continuous distribution may be truncated to an interval $[a, b]$ to define a new density $p_{[a,b]}(x)$ with support $[a, b]$ by setting

$$p_{[a,b]}(x) = \frac{p(x)}{\int_a^b p(u)\,du}.$$

A probability mass function $p(x)$ for a discrete distribution may be truncated to the closed interval $[a, b]$ by

$$p_{[a,b]}(x) = \frac{p(x)}{\sum_{u=a}^{b} p(u)}.$$

Truncating with a lower bound

A probability density function $p(x)$ can be truncated to $[a, \infty]$ by defining

$$p_{[a,\infty]}(x) = \frac{p(x)}{\int_a^\infty p(u)\,du}.$$

A probability mass function $p(x)$ is truncated to $[a, \infty]$ by defining

$$p_{[a,\infty]}(x) = \frac{p(x)}{\sum_{a<=u} p(u)}.$$

Truncating with an upper bound

A probability density function $p(x)$ can be truncated to $[-\infty, b]$ by defining

$$p_{[-\infty,b]}(x) = \frac{p(x)}{\int_{-\infty}^b p(u)\,du}.$$

A probability mass function $p(x)$ is truncated to $[-\infty, b]$ by defining

$$p_{[-\infty,b]}(x) = \frac{p(x)}{\sum_{u<=b} p(u)}.$$

Cumulative distribution functions
Given a probability function $p_X(x)$ for a random variable X, its cumulative distribution function (cdf) $F_X(x)$ is defined to be the probability that $X \leq x$,

$$F_X(x) = \Pr[X \leq x].$$

The upper-case variable X is the random variable whereas the lower-case variable x is just an ordinary bound variable. For continuous random variables, the definition of the cdf works out to

$$F_X(x) \;=\; \int_{-\infty}^{x} p_X(u)\, du,$$

For discrete variables, the cdf is defined to include the upper bound given by the argument,

$$F_X(x) = \sum_{u \leq x} p_X(u).$$

Complementary cumulative distribution functions
The complementary cumulative distribution function (ccdf) in both the continuous and discrete cases is given by

$$F_X^C(x) \;=\; \Pr[X > x] \;=\; 1 - F_X(x).$$

Unlike the cdf, the ccdf is exclusive of the bound, hence the event $X > x$ rather than the cdf's event $X \leq x$.

For continuous distributions, the ccdf works out to

$$F_X^C(x) \;=\; 1 - \int_{-\infty}^{x} p_X(u)\, du \;=\; \int_{x}^{\infty} p_X(u)\, du.$$

The lower boundary can be included in the integration bounds because it is a single point on a line and hence has no probability mass. For the discrete case, the lower bound must be excluded in the summation explicitly by summing over $u > x$,

$$F_X^C(x) \;=\; 1 - \sum_{u \leq x} p_X(u) \;=\; \sum_{u > x} p_X(u).$$

Cumulative distribution functions provide the necessary integral calculations to define truncated distributions. For truncation with lower and upper bounds, the denominator is defined by

$$\int_a^b p(u)\,du = F_X(b) - F_X(a).$$

This allows truncated distributions to be defined as

$$p_{[a,b]}(x) = \frac{p_X(x)}{F_X(b) - F_X(a)}.$$

For discrete distributions, a slightly more complicated form is required to explicitly insert the lower truncation point, which is otherwise excluded from $F_X(b) - F_X(a)$,

$$p_{[a,b]}(x) = \frac{p_X(x)}{F_X(b) - F_X(a) + p_X(a)}.$$

Truncation with lower and upper bounds in Stan

Stan allows probability functions to be truncated. For example, a truncated unit normal distributions restricted to $[-0.5, 2.1]$ can be coded with the following distribution statement.

```
y ~ normal(0, 1) T[-0.5, 2.1];
```

Truncated distributions are translated as an additional term in the accumulated log density function plus error checking to make sure the variate in the distribution statement is within the bounds of the truncation.

In general, the truncation bounds and parameters may be parameters or local variables.

Because the example above involves a continuous distribution, it behaves the same way as the following more verbose form.

```
y ~ normal(0, 1);
if (y < -0.5 || y > 2.1) {
  target += negative_infinity();
} else {
  target += -log_diff_exp(normal_lcdf(2.1 | 0, 1),
                          normal_lcdf(-0.5 | 0, 1));
}
```

Because a Stan program defines a log density function, all calculations are on the log scale. The function `normal_lcdf` is the log of the cumulative normal distribution function and the function `log_diff_exp(a, b)` is a more arithmetically stable form of `log(exp(a) - exp(b))`.

For a discrete distribution, another term is necessary in the denominator to account for the excluded boundary. The truncated discrete distribution

```
y ~ poisson(3.7) T[2, 10];
```

behaves in the same way as the following code.

```
y ~ poisson(3.7);
if (y < 2 || y > 10) {
  target += negative_infinity();
} else {
  target += -log_sum_exp(poisson_lpmf(2 | 3.7),
                         log_diff_exp(poisson_lcdf(10 | 3.7),
                                      poisson_lcdf(2 | 3.7)));
}
```

Recall that `log_sum_exp(a, b)` is just the arithmetically stable form of `log(exp(a) + exp(b))`.

Truncation with lower bounds in Stan

For truncating with only a lower bound, the upper limit is left blank.

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

This truncated distribution statement has the same behavior as the following code.

```
y ~ normal(0, 1);
if (y < -0.5) {
  target += negative_infinity();
} else {
  target += -normal_lccdf(-0.5 | 0, 1);
}
```

The `normal_lccdf` function is the normal complementary cumulative distribution function.

As with lower and upper truncation, the discrete case requires a more complicated denominator to add back in the probability mass for the lower bound. Thus

```
y ~ poisson(3.7) T[2, ];
```

behaves the same way as

```
y ~ poisson(3.7);
if (y < 2) {
  target += negative_infinity();
} else {
  target += -log_sum_exp(poisson_lpmf(2 | 3.7),
                         poisson_lccdf(2 | 3.7));
}
```

Truncation with upper bounds in Stan

To truncate with only an upper bound, the lower bound is left blank. The upper truncated distribution statement

```
y ~ normal(0, 1) T[ , 2.1];
```

produces the same result as the following code.

```
target += normal_lpdf(y | 0, 1);
if (y > 2.1) {
  target += negative_infinity();
} else {
  target += -normal_lcdf(2.1 | 0, 1);
}
```

With only an upper bound, the discrete case does not need a boundary adjustment. The upper-truncated distribution statement

```
y ~ poisson(3.7) T[ , 10];
```

behaves the same way as the following code.

```
y ~ poisson(3.7);
if (y > 10) {
  target += negative_infinity();
} else {
  target += -poisson_lcdf(10 | 3.7);
}
```

Cumulative distributions must be defined
In all cases, the truncation is only well formed if the appropriate log density or mass function and necessary log cumulative distribution functions are defined. Not every distribution built into Stan has log cdf and log ccdfs defined, nor will every user-defined distribution. The discrete probability function documentations describes the available discrete and continuous cumulative distribution functions; most univariate distributions have log cdf and log ccdf functions.

Type constraints on bounds
For continuous distributions, truncation points must be expressions of type `int` or `real`. For discrete distributions, truncation points must be expressions of type `int`.

Variates outside of truncation bounds
For a truncated distribution statement, if the value sampled is not within the bounds specified by the truncation expression, the result is zero probability and the entire statement adds $-\infty$ to the total log probability, which in turn results in the sample being rejected.

Vectorizing truncated distributions
Vectorization of distribution functions with truncation is available if the underlying distribution, `lcdf`, and `lccdf` functions meet the required signatures.

The equivalent code for a vectorized truncation depends on which of the variables are non-scalars (arrays, vectors, etc.):

1. If the variate `y` is the only non-scalar, the result is the same as described in the above sections, but the `lcdf/lccdf` calculation is multiplied by `size(y)`.
2. If the other arguments to the distribution are non-scalars, then the vectorized version of the `lcdf/lccdf` is used. These functions return the sum of their terms, so no multiplication by the size is needed.
3. The exception to the above is when a non-variate is a vector and both a lower and upper bound are specified in the truncation. In this case, a for loop is generated over the elements of the non-scalar arguments. This is required since the `log_diff_exp` of two sums is not the same as the sum of the pairwise `log_diff_exp` operations.

Note that while a lower-and-upper truncated distribution may generate a for-loop internally as part of translating the truncation statement, this is still preferable to manually constructing a loop, since the distribution function itself can still be evaluated in a vectorized manner.

7.6. For loops

Suppose `N` is a variable of type `int`, `y` is a one-dimensional array of type `array[] 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.

Loop variable typing and scope

The type of the loop variable is `int`. Unlike in C++ and similarly to R, this variable must not be declared explicitly.

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.

The scope of the loop variable is limited to the body of the loop.

Order sensitivity and repeated variables

Unlike in BUGS, Stan allows variables to be reassigned. For example, the variable `theta` in the following program is reassigned in each iteration of the loop.

```
for (n in 1:N) {
  theta = inv_logit(alpha + x[n] * beta);
  y[n] ~ bernoulli(theta);
}
```

Such reassignment is not permitted in BUGS. In BUGS, for loops are declarative, defining plates in directed graphical model notation, which can be thought of as repeated substructures in the graphical model. Therefore, it is illegal in BUGS or JAGS to have a for loop that repeatedly reassigns a value to a variable.[4]

[4] A programming idiom in BUGS code simulates a local variable by replacing `theta` in the above example with `theta[n]`, effectively creating `N` different variables, `theta[1]`, ..., `theta[N]`. Of course,

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];
}
```

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 * epsilon;
  epsilon = 0.5 * epsilon;
  y += y * epsilon;
}
```

7.7. Foreach loops

A second form of for loops allows iteration over elements of containers. If `ys` is an expression denoting a container (vector, row vector, matrix, or array) with elements of type `T`, then the following is a well-formed foreach statement.

this is not a hack if the value of `theta[n]` is required for all `n`.

```
for (y in ys) {
  // ... do something with y ...
}
```

The order in which elements of ys are visited is defined for container types as follows.

- vector, row_vector: elements visited in order, y is of type double
- matrix: elements visited in column-major order, y is of type double
- array[] T: elements visited in order, y is of type T.

Consequently, if ys is a two dimensional array array[,] real, y will be a one-dimensional array of real values (type array[] real). If 'ysis a matrix, thenywill be a real value (typereal'). To loop over all values of a two-dimensional array using foreach statements would require a doubly-nested loop,

```
array[2, 3] real yss;
for (ys in yss) {
  for (y in ys) {
    // ... do something with y ...
  }
}
```

whereas a matrix can be looped over in one foreach statement

```
matrix[2, 3] yss;
for (y in yss) {
   // ... do something with y...
}
```

In both cases, the loop variable y is of type real. The elements of the matrix are visited in column-major order (e.g.,y[1, 1],y[2, 1],y[1, 2], ...,y[2, 3]), whereas the elements of the two-dimensional array are visited in row-major order (e.g.,y[1, 1],y[1, 2],y[1, 3],y[2, 1], ...,y[2, 3]').

7.8. 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 an 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 evaluate to a non-zero value and there is a final else clause, its statement is executed.

7.9. 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 expression and the body can be any statement (or sequence of statements in curly braces).

Evaluation of a while loop starts by evaluating the condition. If the condition evaluates to a false (zero) value, the execution of the loop terminates and control moves to the position after the loop. If the loop's condition evaluates to a true (non-zero) value, the body statement is executed, then the whole loop is executed again. Thus the loop is continually executed as long as the condition evaluates to a true value.

The rest of the body of a while loop may be skipped using a `continue`. The loop will be exited with a `break` statement. See the section on continue and break statements for more details.

7.10. 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 listed in the types table under "local".

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 = 2 * m;
     sum += n;
  }
  for (n in 1:N) {
    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.

7.11. Break and continue statements

The one-token statements `continue` and `break` may be used within loops to alter control flow; `continue` causes the next iteration of the loop to run immediately, whereas `break` terminates the loop and causes execution to resume after the loop. Both control structures must appear in loops. Both `break` and `continue` scope to the most deeply nested loop, but pass through non-loop statements.

Although these control statements may seem undesirable because of their goto-like behavior, their judicious use can greatly improve readability by reducing the level of nesting or eliminating bookkeeping inside loops.

Break statements

When a `break` statement is executed, the most deeply nested loop currently being executed is ended and execution picks up with the next statement after the loop. For example, consider the following program:

```
while (1) {
  if (n < 0) {
    break;
  }
  foo(n);
  n = n - 1;
}
```

The `while~(1)` loop is a "forever" loop, because `1` is the true value, so the test always succeeds. Within the loop, if the value of `n` is less than 0, the loop terminates,

otherwise it executes `foo(n)` and then decrements `n`. The statement above does exactly the same thing as

```
while (n >= 0) {
  foo(n);
  n = n - 1;
}
```

This case is simply illustrative of the behavior; it is not a case where a `break` simplifies the loop.

Continue statements

The `continue` statement ends the current operation of the loop and returns to the condition at the top of the loop. Such loops are typically used to exclude some values from calculations. For example, we could use the following loop to sum the positive values in the array `x`,

```
real sum;
sum = 0;
for (n in 1:size(x)) {
  if (x[n] <= 0) {
    continue;
  }
  sum += x[n];
}
```

When the continue statement is executed, control jumps back to the conditional part of the loop. With while and for loops, this causes control to return to the conditional of the loop. With for loops, this advances the loop variable, so the the above program will not go into an infinite loop when faced with an `x[n]` less than zero. Thus the above program could be rewritten with deeper nesting by reversing the conditional,

```
real sum;
sum = 0;
for (n in 1:size(x)) {
  if (x[n] > 0) {
    sum += x[n];
  }
}
```

While the latter form may seem more readable in this simple case, the former has

the main line of execution nested one level less deep. Instead, the conditional at the top finds cases to exclude and doesn't require the same level of nesting for code that's not excluded. When there are several such exclusion conditions, the break or continue versions tend to be much easier to read.

Breaking and continuing nested loops

If there is a loop nested within a loop, a `break` or `continue` statement only breaks out of the inner loop. So

```
while (cond1) {
  // ...
  while (cond2) {
    // ...
    if (cond3) {
      break;
    }
    // ...
  }
  // execution continues here after break
  // ...
}
```

If the break is triggered by `cond3` being true, execution will continue after the nested loop.

As with break statements, continue statements go back to the top of the most deeply nested loop in which the `continue` appears.

Although break and continue must appear within loops, they may appear in nested statements within loops, such as within the conditionals shown above or within nested statements. The break and continue statements jump past any control structure other than while-loops and for-loops.

7.12. Print statements

Stan provides print statements that can print literal strings and the values of expressions. Print statements accept any number of arguments. Consider the following for-each statement with a print statement in its body.

```
for (n in 1:N) { print("loop iteration: ", n); ... }
```

The print statement will execute every time the body of the loop does. Each time the loop body is executed, it will print the string "loop iteration:" (with the trailing

space), followed by the value of the expression `n`, followed by a new line.

Print content

The text printed by a print statement varies based on its content. A literal (i.e., quoted) string in a print statement always prints exactly that string (without the quotes). Expressions in print statements result in the value of the expression being printed. But how the value of the expression is formatted will depend on its type.

Printing a simple `real` or `int` typed variable always prints the variable's value.[5]

For array, vector, and matrix variables, the print format uses brackets. For example, a 3-vector will print as

```
[1, 2, 3]
```

and a 2×3-matrix as

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

Complex numbers print as pairs. For example, the pair of statements

```
complex z = to_complex(1.2, -3.5);
print(z)
```

will print as `(1.2,-3.5)`, with no space after the comma or within the parentheses.

Printing a more readable version of arrays or matrices can be done with loops. An example is the print statement in the following transformed data block.

```
transformed data {
  matrix[2, 2] u;
  u[1, 1] = 1.0;   u[1, 2] = 4.0;
  u[2, 1] = 9.0;   u[2, 2] = 16.0;
  for (n in 1:2) {
    print("u[", n, "] = ", u[n]);
  }
}
```

This print statement executes twice, printing the following two lines of output.

```
u[1] = [1, 4]
u[2] = [9, 16]
```

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

Non-void input

The input type to a print function cannot be void. In particular, it can't be the result of a user-defined void function. All other types are allowed as arguments to the print function.

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 any byte sequence, with the exception of the double quote character.

The Stan interfaces preserve the byte sequences which they receive. The encoding of these byte sequences as characters and their rendering as glyphs will be handled by whatever display mechanism is being used to monitor Stan's output (e.g., a terminal, a Jupyter notebook, RStudio, etc.). Stan does not enforce a character encoding for strings, and no attempt is made to validate the bytes as legal ASCII, UTF-8, etc.

Debug by `print`

Because Stan is an imperative language, print statements can be very useful for debugging. They can be used to display the values of variables or expressions at various points in the execution of a program. They are particularly useful for spotting problematic not-a-number of infinite values, both of which will be printed.

It is particularly useful to print the value of the target log density accumulator (through the `target()` function), as in the following example.

```
vector[2] y;
y[1] = 1;
print("log density before =", target());
y ~ normal(0,1);  // bug!  y[2] not defined
print("log density after =", target());
```

The example has a bug in that `y[2]` is not defined before the vector `y` is used in the distribution statement. By printing the value of the log probability accumulator before and after each distribution statement, it's possible to isolate where the log probability becomes ill-defined (i.e., becomes not-a-number).

Note that print statements may not always be displayed immediately, but rather at the end of an operation (e.g., leapfrog step). As such, some issues such as infinite loops are difficult to debug effectively with this technique.

7.13. Reject statements

The Stan `reject` statement provides a mechanism to report errors or problematic values encountered during program execution and either halt processing or reject iterations.

Like the `print` statement, the reject statement accepts any number of quoted string literals or Stan expressions as arguments.

Reject statements are typically embedded in a conditional statement in order to detect variables in illegal states. For example, the following code handles the case where a variable `x`'s value is negative.

```
if (x < 0) {
  reject("x must not be negative; found x=", x);
}
```

Behavior of reject statements

Reject statements have the same behavior as exceptions thrown by built-in Stan functions. For example, the `normal_lpdf` function raises an exception if the input scale is not positive and finite. The effect of a reject statement depends on the program block in which the rejection occurs.

In all cases of rejection, the interface accessing the Stan program should print the arguments to the reject statement.

Rejections in functions

Rejections in user-defined functions are just passed to the calling function or program block. Reject statements can be used in functions to validate the function arguments, allowing user-defined functions to fully emulate built-in function behavior. It is better to find out earlier rather than later when there is a problem.

Fatal exception contexts

Rejections are fatal in the transformed data block. This is because if initialization fails there is no way to recover values, so the algorithm will not begin execution.

Reject statements placed in the transformed data block can be used to validate both the data and transformed data (if any). This allows more complicated constraints to be enforced that can be specified with Stan's constrained variable declarations.

Fatal errors in other blocks may also be signaled by use of the `fatal_error` statement.

Recoverable rejection contexts

Rejections in the transformed parameters and model blocks are not in and of themselves instantly fatal. The result has the same effect as assigning a $-\infty$ log probability, which causes rejection of the current proposal in MCMC samplers and adjustment of search parameters in optimization.

If the log probability function results in a rejection every time it is called, the containing application (MCMC sampler or optimization) should diagnose this problem and terminate with an appropriate error message. To aid in diagnosing problems, the message for each reject statement will be printed as a result of executing it.

Rejection is not for constraints

Rejection should be used for error handling, not defining arbitrary constraints. Consider the following errorful Stan program.

```
parameters {
  real a;
  real<lower=a> b;
  real<lower=a, upper=b> theta;
  // ...
}
model {
  // **wrong** needs explicit truncation
  theta ~ normal(0, 1);
  // ...
}
```

This program is wrong because its truncation bounds on `theta` depend on parameters, and thus need to be accounted for using an explicit truncation on the distribution. This is the right way to do it.

```
  theta ~ normal(0, 1) T[a, b];
```

The conceptual issue is that the prior does not integrate to one over the admissible parameter space; it integrates to one over all real numbers and integrates to something less than one over $[a, b]$; in these simple univariate cases, we can overcome that with the `T[ , ]` notation, which essentially divides by whatever the prior integrates to over $[a, b]$.

This problem is exactly the same problem as you would get using reject statements

to enforce complicated inequalities on multivariate functions. In this case, it is wrong to try to deal with truncation through constraints.

```
if (theta < a || theta > b) {
  reject("theta not in (a, b)");
}
// still **wrong**, needs T[a,b]
theta ~ normal(0, 1);
```

In this case, the prior integrates to something less than one over the region of the parameter space where the complicated inequalities are satisfied. But we don't generally know what value the prior integrates to, so we can't increment the log probability function to compensate.

Even if this adjustment to a proper probability model may seem minor in particular models where the amount of truncated posterior density is negligible or constant, we can't sample from that truncated posterior efficiently. Programs need to use one-to-one mappings that guarantee the constraints are satisfied and only use reject statements to raise errors or help with debugging.

7.14. Fatal error statements

The Stan `fatal_error` statement provides a mechanism to report errors or problematic values encountered during program execution and uniformly halt processing.

Like the `print` or `reject` statements, the fatal error statement accepts any number of quoted string literals or Stan expressions as arguments.

The fatal error may be used to signal an unrecoverable error in blocks where `reject` leads to the algorithm attempting to try again, such as the `model` block.

8. 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.

8.1. Overview of Stan's program blocks

The full set of named program blocks is exemplified in the following skeletal Stan program.

```
functions {
  // ... function declarations and definitions ...
}
data {
  // ... declarations ...
}
transformed data {
   // ... declarations ... statements ...
}
parameters {
   // ... declarations ...
}
transformed parameters {
   // ... declarations ... statements ...
}
model {
   // ... declarations ... statements ...
}
generated quantities {
   // ... declarations ... statements ...
}
```

The function-definition block contains user-defined functions. The data block declares the required data for the model. The transformed data block allows the definition of constants and transforms of the data. The parameters block declares the model's parameters — the unconstrained version of the parameters is what's sampled or optimized. The transformed parameters block allows variables to be defined in terms of data and parameters that may be used later and will be saved.

The model block is where the log probability function is defined. The generated quantities block allows derived quantities based on parameters, data, and optionally (pseudo) random number generation.

Optionality and ordering

All of the blocks are optional. A consequence of this is that the empty string is a valid Stan program, although it will trigger a warning message from the Stan compiler. The Stan program 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. The exception to this rule is that variables declared in the model block are always local to the model block and may not be accessed in the generated quantities block; to make a variable accessible in the model and generated quantities block, it must be declared as a transformed parameter.

Variables declared as function parameters have scope only within that function definition's body, and may not be assigned to (they are constant).

Function scope

Functions defined in the function block may be used in any appropriate block. Most functions can be used in any block and applied to a mixture of parameters and data (including constants or program literals).

Random-number-generating functions are restricted to transformed data and generated quantities blocks, and within user-defined functions ending in `_rng`; such functions are suffixed with `_rng`. Log-probability modifying functions to blocks where the log probability accumulator is in scope (transformed parameters and model); such functions are suffixed with `_lp`.

Density functions defined in the program may be used in distribution statements.

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 draws, 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 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 the block actions table.

Block Actions Table. *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.*

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

block	statement	action / period
`data`	no	read / chain
`transformed data`	yes	evaluate / chain
`parameters`	no	inv. transform, Jacobian / leapfrog inv. transform, write / sample
`transformed parameters`	yes	evaluate / leapfrog write / sample
`model`	yes	evaluate / leapfrog step
`generated quantities`	yes	eval / sample write / sample
(initialization)	n/a	read, transform / chain

Variable Declaration Table. *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.*

param depend	in target	save	declare in
+	+	+	`transformed parameters`
+	+	-	`model` (local)
+	-	+	`generated quantities`
+	-	-	`generated quantities` (local)
-	+	+	`transformed data` *and* `generated quantities`
-	+	-	`transformed data`
-	-	+	`generated quantities`
-	-	-	`transformed data` (local)

Another way to look at the variables is in terms of their function. To decide which variable to use, consult the charts in the variable declaration table. 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.

[2]It is possible to print a variable every iteration that does not depend on parameters—just define it (or redefine it if it is transformed data) in the `generated quantities` block.

8.2. Statistical variable taxonomy

Statistical Variable Taxonomy Table. *Variables of the kind indicated in the left column must be declared in one of the blocks declared in the right column.*

variable kind	declaration block
constants	`data`, `transformed data`
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`
derived quantities	`transformed data`, `transformed parameters`, `generated quantities`
loop indices	loop statement

Page 366 of (Gelman and Hill 2007) provides a taxonomy of the kinds of variables used in Bayesian models. The table of kinds of variables 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
  array[N] real y;          // modeled data
  real mu_mu;               // config. unmodeled param
  real<lower=0> sigma_mu;   // config. unmodeled param
}
transformed data {
  real<lower=0> alpha;      // const. unmodeled param
  real<lower=0> beta;       // const. unmodeled param
  alpha = 0.1;
  beta = 0.1;
```

```
}
parameters {
  real mu_y;                // modeled param
  real<lower=0> tau_y;      // modeled param
}
transformed parameters {
  real<lower=0> sigma_y;    // derived quantity (param)
  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` is an array of modeled data. 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.

8.3. 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 draws 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 or data in memory depends on the interface; see the user's guides and interface documentation for PyStan, RStan, and CmdStan for details.

Statements

The `data` block does not allow statements.

[3] With multiple threads, or even running chains sequentially in a single thread, data could be read only once per set of chains. Stan was designed to be thread safe and future versions will provide a multithreading option for Markov chains.

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.

8.4. 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. 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.

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

Restriction on operations in `transformed data`

The statements in the transformed data block are designed to be executed once and have a deterministic result. Therefore, log probability is not accumulated and distribution statements may not be used.

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.

8.5. 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 the variable transforms chapter.

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 draws

In the basic Stan compiled program, there is a file to which the values of variables are written for each draw. The constrained versions of the variables are written in the order they are defined in the `parameters` block. In order to do this, the transformed parameter, model, and generated quantities statements must also be executed.

8.6. 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 draws.

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.

Constraints are for error checking

Like the constraints on data, the constraints on transformed parameters is meant to catch programming errors as well as convey programmer intent. They are not automatically transformed in such a way as to be satisfied. What will happen if a transformed parameter does not match its constraint is that the current parameter values will be rejected. This can cause Stan's algorithms to hang or to devolve to random walks. It is not intended to be a way to enforce ad hoc constraints in Stan programs. See the section on reject statements for further discussion of the behavior of reject statements.

8.7. 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 (distribution notation) statements are allowed. These are typically used when programming in the BUGS idiom to define the probability model.

8.8. 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.

Parameter estimates, predictions, statistics, and event probabilities calculated directly using plug-in estimates. Stan automatically provides full Bayesian inference by producing draws from the posterior distribution of any calculated event probabilities, predictions, or statistics.

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. Variables declared in nested blocks are local variables, not generated quantities, and thus won't be printed. For example:

```
generated quantities {
  int a; // added to the output

  {
    int b; // not added to the output
  }
}
```

9. User-Defined Functions

Stan allows users to define their own functions. The basic syntax is a simplified version of that used in C and C++. This chapter specifies how functions are declared, defined, and used in Stan.

9.1. Function-definition block

User-defined functions appear in a special function-definition block before all of the other program blocks.

```
functions {
   // ... function declarations and definitions ...
}
data {
  // ...
```

Function definitions and declarations may appear in any order. Forward declarations are allowed but not required.

9.2. Function names

The rules for function naming and function-argument naming are the same as for other variables; see the section on variables for more information on valid identifiers. For example,

```
real foo(real mu, real sigma);
```

declares a function named `foo` with two argument variables of types `real` and `real`. The arguments are named `mu` and `sigma`, but that is not part of the declaration.

Function overloading

Multiple user-defined functions may have the same name if they have different sequences of argument types. This is known as function overloading.

For example, the following two functions are both defined with the name `add_up`

```
real add_up(real a, real b){
  return a + b;
}
```

```
real add_up(real a, real b, real c){
  return a + b + c;
}
```

The return types of overloaded functions do not need to be the same. One could define an additional `add_up` function as follows

```
int add_up(int a, int b){
  return a + b;
}
```

That being said, functions may **not** use the same name if their signature *only* differs by the return type.

For example, the following is not permitted

```
// illegal
real baz(int x);
int baz(int x);
```

Function names used in the Stan standard library may be overloaded by user-defined functions. Exceptions to this are the `reduce_sum` family of functions and ODE integrators, which cannot be overloaded.

9.3. Calling functions

All function arguments are mandatory—there are no default values.

Functions as expressions

Functions with non-void return types are called just like any other built-in function in Stan—they are applied to appropriately typed arguments to produce an expression, which has a value when executed.

Functions as statements

Functions with void return types may be applied to arguments and used as statements.qmd. These act like distribution statements or print statements. Such uses are only appropriate for functions that act through side effects, such as incrementing the log probability accumulator, printing, or raising exceptions.

Resolving overloads

Overloaded functions alongside type promotion can result in situations where there are multiple valid interpretations of a function call. Stan requires that there be a unique signature which minimizes the number of promotions required.

Consider the following two overloaded functions

```
real foo(int a, real b);
real foo(real a, int b);
```

These functions do **not** have a unique minimum when called with two integer arguments `foo(1,2)`, and therefore cannot be called as such.

Promotion of integers to complex numbers is considered as two separate promotions, one from `int` to `real` and a second from `real` to `complex`. Consider the following functions with `real` and `complex` signatures

```
real bar(real x);
real bar(complex z);
```

A call `bar(5)` with an integer argument will be resolved to `bar(real)` because it only requires a single promotion, whereas the promotion to a complex number requires two promotions.

Argument promotion

The rules for calling functions work the same way as assignment as far as promotion goes. This means that we can promote arguments to the type expected by function arguments. For example, the following will work.

```
real foo(real x) { return ... };
...
int a = 5;
real b = foo(a); // a promoted to type real
```

In addition to promoting `int` to `real`, Stan also promotes `real` to `complex`, and by transitivity, `int` to `complex`. This also works for containers, so an array of `int` may be assigned to an array of `real` of the same shape. And we can also promote `vector` to `complex_vector` and similarly for row vectors and matrices.

Probability functions in distribution statements

Functions whose name ends in `_lpdf` or `_lpmf` (log density and mass functions) may be used as probability functions and may be used in place of parameterized distributions on the right side of statements.qmd#distribution-statements.section.

Restrictions on placement

Functions of certain types are restricted on scope of usage. Functions whose names end in `_lp` assume access to the log probability accumulator and are only available in the transformed parameter and model blocks.

Functions whose names end in `_rng` assume access to the random number generator and may only be used within the generated quantities block, transformed data block, and within user-defined functions ending in `_rng`.

Functions whose names end in `_lpdf` and `_lpmf` can be used anywhere. However, `_lupdf` and `_lupmf` functions can only be used in the model block or user-defined probability functions.

See the section on function bodies for more information on these special types of function.

9.4. Argument types and qualifiers

Stan's functions all have declared types for both arguments and returned value. As with built-in functions, user-defined functions are only declared for base argument type and dimensionality. This requires a different syntax than for declaring other variables. The choice of language was made so that return types and argument types could use the same declaration syntax.

The type `void` may not be used as an argument type, only a return type for a function with side effects.

Base variable type declaration

The base variable types are `integer`, `real`, `complex`, `vector`, `row_vector`, and `matrix`. No lower-bound or upper-bound constraints are allowed (e.g., `real<lower=0>` is illegal). Specialized constrained types are also not allowed (e.g., `simplex` is illegal).

Tuple types of the form `tuple(T1, ..., TN)` are also allowed, with all of the types `T1` to `TN` being function argument types (i.e., no constraints and no sizes).

Dimensionality declaration

Arguments and return types may be arrays, and these are indicated with optional brackets and commas as would be used for indexing. For example, `int` denotes a single integer argument or return, whereas `array[] real` indicates a one-dimensional array of reals, `array[,] real` a two-dimensional array and `array[,,] real` a three-dimensional array; whitespace is optional, as usual.

The dimensions for vectors and matrices are not included, so that `matrix` is the type of a single matrix argument or return type. Thus if a variable is declared as `matrix a`, then `a` has two indexing dimensions, so that `a[1]` is a row vector and `a[1, 1]` a real value. Matrices implicitly have two indexing dimensions. The type declaration `matrix[ , ] b` specifies that `b` is a two-dimensional array of matrices, for a total

of four indexing dimensions, with `b[1, 1, 1, 1]` picking out a real value.

Dimensionality checks and exceptions

Function argument and return types are not themselves checked for dimensionality. A matrix of any size may be passed in as a matrix argument. Nevertheless, a user-defined function might call a function (such as a multivariate normal density) that itself does dimensionality checks.

Dimensions of function return values will be checked if they're assigned to a previously declared variable. They may also be checked if they are used as the argument to a function.

Any errors raised by calls to functions inside user functions or return type mismatches are simply passed on; this typically results in a warning message and rejection of a proposal during sampling or optimization.

Data-only qualifiers

Some of Stan's built-in functions, like the differential equation solvers, have arguments that must be data. Such data-only arguments must be expressions involving only data, transformed data, and generated quantity variables.

In user-defined functions, the qualifier `data` may be placed before an argument type declaration to indicate that the argument must be data only. For example,

```
real foo(data real x) {
  return x^2;
}
```

requires the argument `x` to be data only.

Declaring an argument data only allows type inference to proceed in the body of the function so that, for example, the variable may be used as a data-only argument to a built-in function.

9.5. Function bodies

The body of a function is between an open curly brace (`{`) and close curly brace (`}`). The body may contain local variable declarations at the top of the function body's block and these scope the same way as local variables used in any other statement block.

Any user-defined function may be used in the function body regardless of the order in which the function definitions appear in the file. Self-recursive and mutually recursive functions are possible without any additional declarations.

The only restrictions on statements in function bodies are external, and determine whether the log probability accumulator or random number generators are available; see the rest of this section for details.

Random number generating functions

Functions that call random number generating functions in their bodies must have a name that ends in `_rng`; attempts to use random-number generators in other functions lead to a compile-time error.

Like other random number generating functions, user-defined functions with names that end in `_rng` may be used only in the generated quantities block and transformed data block, or within the bodies of user-defined functions ending in `_rng`. An attempt to use such a function elsewhere results in a compile-time error.

Log probability access in functions

Functions that include statements.qmd#distribution-statements.section or statements.qmd#increment-log-prob.section must have a name that ends in `_lp`. Attempts to use distribution statements or increment log probability statements in other functions lead to a compile-time error.

Like the target log density increment statement and distribution statements, user-defined functions with names that end in `_lp` may only be used in blocks where the log probability accumulator is accessible, namely the transformed parameters and model blocks. An attempt to use such a function elsewhere results in a compile-time error.

Defining probability functions for distribution statements

Functions whose names end in `_lpdf` and `_lpmf` (density and mass functions) can be used as probability functions in distribution statements. As with the built-in functions, the first argument will appear on the left of the distribution statement operator (~) in the distribution statement and the other arguments follow. For example, suppose a function returning the log of the density of `y` given parameter `theta` allows the use of the distribution statement is defined as follows.

```
real foo_lpdf(real y, vector theta) { ... }
```

Note that for function definitions, the comma is used rather than the vertical bar.

For every custom `_lpdf` and `_lpmf` defined there is a corresponding `_lupdf` and `_lupmf` defined automatically. The `_lupdf` and `_lupmf` versions of the functions cannot be defined directly (to do so will produce an error). The difference in the `_lpdf` and `_lpmf` and the corresponding `_lupdf` and `_lupmf` functions is that if any other unnormalized density functions are used inside the user-defined function, the

_lpdf and _lpmf forms of the user-defined function will change these densities to be normalized. The _lupdf and _lupmf forms of the user-defined functions will instead allow other unnormalized density functions to drop additive constants.

The distribution statement shorthand

```
z ~ foo(phi);
```

will have the same effect as incrementing the target with the log of the unnormalized density:

```
target += foo_lupdf(z | phi);
```

Other _lupdf and _lupmf functions used in the definition of foo_lpdf will drop additive constants when foo_lupdf is called and will not drop additive constants when foo_lpdf is called.

If there are _lupdf and _lupmf functions used inside the following call to foo_lpdf, they will be forced to normalize (return the equivalent of their _lpdf and _lpmf forms):

```
target += foo_lpdf(z | phi);
```

If there are no _lupdf or _lupmf functions used in the definition of foo_lpdf, then there will be no difference between a foo_lpdf or foo_lupdf call.

The unnormalized _lupdf and _lupmf functions can only be used in the model block or in user-defined probability functions (those ending in _lpdf or _lpmf).

The same syntax and shorthand that works for _lpdf also works for log probability mass functions with suffixes _lpmf.

A function that is going to be accessed as distributions must return the log of the density or mass function it defines.

9.6. Parameters are constant

Within function definition bodies, the parameters may be used like any other variable. But the parameters are constant in the sense that they can't be assigned to (i.e., can't appear on the left side of an assignment (=) statement). In other words, their value remains constant throughout the function body. Attempting to assign a value to a function parameter value will raise a compile-time error.[1]

[1]Despite being declared constant and appearing to have a pass-by-value syntax in Stan, the implementation of the language passes function arguments by constant reference in C++.

Local variables may be declared at the top of the function block and scope as usual.

9.7. Return value

Non-void functions must have a return statement that returns an appropriately typed expression. If the expression in a return statement does not have the same type as the return type declared for the function, a compile-time error is raised.

Void functions may use `return` only without an argument, but return statements are not mandatory.

Return guarantee required

Unlike C++, Stan enforces a syntactic guarantee for non-void functions that ensures control will leave a non-void function through an appropriately typed return statement or because an exception is raised in the execution of the function. To enforce this condition, functions must have a return statement as the last statement in their body. This notion of last is defined recursively in terms of statements that qualify as bodies for functions. The base case is that

- a return statement qualifies,

and the recursive cases are that

- a sequence of statements qualifies if its last statement qualifies,
- a for loop or while loop qualifies if its body qualifies, and
- a conditional statement qualifies if it has a default else clause and all of its body statements qualify.

An exception is made for "obviously infinite" loops like `while (1)`, which contain a `return` statement and no `break` statements. The only way to exit such a loop is to return, so they are considered as returning statements.

These rules disqualify

```
real foo(real x) {
  if (x > 2) {
    return 1.0;
  } else if (x <= 2) {
    return -1.0;
  }
}
```

because there is no default `else` clause, and disqualify

```
real foo(real x) {
  real y;
  y = x;
  while (x < 10) {
    if (x > 0) {
      return x;
    }
    y = x / 2;
  }
}
```

because the return statement is not the last statement in the while loop. A bogus dummy return could be placed after the while loop in this case. The rules for returns allow

```
real log_fancy(real x) {
  if (x < 1e-30) {
    return x;
  } else if (x < 1e-14) {
    return x * x;
  } else {
    return log(x);
  }
}
```

because there's a default else clause and each condition body has return as its final statement.

9.8. Void Functions as Statements

Void functions

A function can be declared without a return value by using `void` in place of a return type. Note that the type `void` may only be used as a return type—arguments may not be declared to be of type `void`.

Usage as statement

A void function may be used as a statement.

Because there is no return, such a usage is only for side effects, such as incrementing the log probability function, printing, or raising an error.

Special return statements

In a return statement within a void function's definition, the `return` keyword is followed immediately by a semicolon (`;`) rather than by the expression whose value is returned.

9.9. Declarations

Stan supports forward declarations, which look like function definitions without bodies. For example,

```
real unit_normal_lpdf(real y);
```

declares a function named `unit_normal_lpdf` that consumes a single real-valued input and produces a real-valued output. Declaring a function without a definition is only really useful when using an extension which supplies the definition in C++ rather than in the Stan code itself. How exactly this can be accomplished will differ depending on your Stan interface.

A function definition with a body simultaneously declares and defines the named function, as in

```
real unit_normal_lpdf(real y) {
  return -0.5 * square(y);
}
```

A function can be declared and (perhaps separately) defined at most once. However, functions with different argument types are considered distinct even if they have the same name; see the section on function overloading.

10. Constraint 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 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. For examples of how to declare and define these variables in a Stan program, see section Variable declaration.

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.

10.1. Limitations due to finite accuracy presentation

In this section the transformations are described mathematically. There are two cases where the observed behavior can be different from the exact arithmetic: - Stan's arithmetic is implemented using double-precision floating-point arithmetic, which may cause computation to behave differently than mathematics. For example, lower bound constraint is defined with logarithm constraint which mathematically excludes the lower bound, but if the closest floating-point number for the inverse transformed value is the boundary, then the value is rounded to the boundary. This may cause unexpected warnings or errors, if in other parts of the code the boundary value is invalid. For example, we may observe floating-point value 0 for a variance parameter that has been declared to be larger than 0. See more about Floating point Arithmetic in Stan user's guide). - CmdStan stores the output to CSV files with 6 significant digits accuracy by default, but the constraints are checked with 8 decimal digit accuracy. Due to this, there can be errors if CSV output is further used, for example, to run generated quantities. For example, simplex constraint requires the values to sum up to 1, but when writing the values to CSV they are rounded to 6 significant digits and the sum of those rounded values can be smaller or larger

than 1 by more than 8 decimal digits. The solution for CmdStan is to increase the number of significant digits stored as discussed in CmdStan Command-Line Interface Overview.

10.2. Changes of variables

The support of a random variable X with density $p_X(x)$ is that subset of values for which it has non-zero density,

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

If f is a total function defined on the support of X, then $Y = f(X)$ is a new random variable. This section shows how to compute the probability density function of Y for well-behaved transforms f. The rest of the chapter details the transforms used by Stan.

Univariate changes of variables

Suppose X is one dimensional and $f : \text{supp}(X) \to \mathbb{R}$ is a one-to-one, monotonic 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.

10.3. Lower bounded scalar

Stan uses a logarithmic transform for lower and upper bounds.

Lower bound transform

If a variable X is declared to have lower bound a, it is transformed to an unbounded variable Y, where

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

Lower bound inverse transform

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

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

Absolute derivative of the lower bound inverse transform
The absolute derivative of the inverse transform is

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

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

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

10.4. Upper bounded scalar

Stan uses a negated logarithmic transform for upper bounds.

Upper bound transform
If a variable X is declared to have an upper bound b, it is transformed to the unbounded variable Y by

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

Upper bound inverse transform
The inverse of the upper bound transform converts the unbounded variable Y to the variable X bounded above by b through

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

Absolute derivative of the upper bound inverse transform
The absolute derivative of the inverse of the upper bound transform is

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

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

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

10.5. Lower and upper bounded scalar

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

Log odds and the logistic sigmoid

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

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

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

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

The derivative of the logistic sigmoid is

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

Lower and upper bounds transform

For variables constrained to be in the open interval (a, b), Stan uses a scaled and translated log-odds transform. If variable X is declared to have lower bound a and upper bound b, then it is transformed to a new variable Y, where

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

Lower and upper bounds inverse transform

The inverse of this transform is

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

Absolute derivative of the lower and upper bounds inverse transform

The absolute derivative of the inverse transform is given by

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

Therefore, the density of the transformed variable Y is

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

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

10.6. Affinely transformed scalar

Stan uses an affine transform to be able to specify parameters with a given offset and multiplier.

Affine transform

For variables with expected offset μ and/or (positive) multiplier σ, Stan uses an affine transform. Such a variable X is transformed to a new variable Y, where

$$Y = \frac{X - \mu}{\sigma}.$$

The default value for the offset μ is 0 and for the multiplier σ is 1 in case not both are specified.

Affine inverse transform

The inverse of this transform is

$$X = \mu + \sigma \cdot Y.$$

Absolute derivative of the affine inverse transform

The absolute derivative of the affine inverse transform is

$$\left| \frac{d}{dy} \left(\mu + \sigma \cdot y \right) \right| = \sigma.$$

Therefore, the density of the transformed variable Y is

$$p_Y(y) = p_X(\mu + \sigma \cdot y) \cdot \sigma.$$

For an example of how to code this in Stan, see section Affinely Transformed Real.

10.7. 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.

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

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

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

Ordered transform

Stan's transform follows the constraint directly. It maps an increasing vector $x \in \mathbb{R}^K$ to an unconstrained vector $y \in \mathbb{R}^K$ by setting

$$y_k = \begin{cases} x_1 & \text{if } k = 1, \text{ and} \\ \log(x_k - x_{k-1}) & \text{if } 1 < k \leq K. \end{cases}$$

Ordered inverse transform

The inverse transform for an unconstrained $y \in \mathbb{R}^K$ to an ordered sequence $x \in \mathbb{R}^K$ is defined by the recursion

$$x_k = \begin{cases} y_1 & \text{if } k = 1, \text{ and} \\ x_{k-1} + \exp(y_k) & \text{if } 1 < k \leq K. \end{cases}$$

x_k can also be expressed iteratively as

$$x_k = y_1 + \sum_{k'=2}^{k} \exp(y_{k'}).$$

Absolute Jacobian determinant of the ordered inverse transform

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

$$J_{k,k} = \begin{cases} 1 & \text{if } k = 1, \text{ and} \\ \exp(y_k) & \text{if } 1 < k \leq K. \end{cases}$$

Because J is triangular, the absolute Jacobian determinant is

$$|\det J| = \left| \prod_{k=1}^{K} J_{k,k} \right| = \prod_{k=2}^{K} \exp(y_k).$$

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

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

10.8. 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 $x_1, x_2, \ldots, x_{K-1}$, with

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

Unit simplex inverse transform

Stan's unit simplex inverse transform may be understood using the following stick-breaking metaphor.[1]

1. Take a stick of unit length (i.e., length 1).
2. Break a piece off and label it as x_1, and set it aside, keeping what's left.
3. Next, break a piece off what's left, label it x_2, and set it aside, keeping what's left.
4. Continue breaking off pieces of what's left, labeling them, and setting them aside for pieces $x_3, \ldots, x_{K-1}$.
5. Label what's left x_K.

The resulting vector $x = [x_1, \ldots, x_K]^\top$ is a unit simplex because each piece has non-negative length and the sum of the stick lengths is one by construction.

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 $\log\left(\frac{1}{K-k}\right)$ (*i.e.*, $\text{logit}\left(\frac{1}{K-k+1}\right)$) in the above definition adjusts the

[1] For an alternative derivation of the same transform using hyperspherical coordinates, see (Betancourt 2010).

transform so that a zero vector y is mapped to the simplex $x = (1/K, \ldots, 1/K)$. For instance, if $y_1 = 0$, then $z_1 = 1/K$; if $y_2 = 0$, then $z_2 = 1/(K-1)$; and if $y_{K-1} = 0$, then $z_{K-1} = 1/2$.

The break proportions z are applied to determine the stick sizes and resulting value of x_k for $1 \leq k < K$ 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 $x_1, \ldots, x_{k-1}$.

Because the Jacobian J of f^{-1} is lower triangular and positive, its absolute determinant reduces to

$$|\det J| \;=\; \prod_{k=1}^{K-1} J_{k,k} \;=\; \prod_{k=1}^{K-1} z_k\,(1-z_k)\;\left(1-\sum_{k'=1}^{k-1} x_{k'}\right).$$

Thus the transformed variable $Y = f(X)$ has a density given by

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

Even though it is expressed in terms of intermediate values z_k, this expression still looks more complex than it is. The exponential function need only be evaluated once for each unconstrained parameter y_k; everything else is just basic arithmetic that can be computed incrementally along with the transform.

Unit simplex transform

The transform $Y = f(X)$ can be derived by reversing the stages of the inverse transform. Working backwards, given the break proportions z, y is defined elementwise by

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

The break proportions z_k are defined to be the ratio of x_k to the length of stick left after the first $k-1$ pieces have been broken off,

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

10.9. Unit vector

An n-dimensional vector $x \in \mathbb{R}^n$ is said to be a unit vector if it has unit Euclidean length, so that

$$\|x\| \;=\; \sqrt{x^\top x} \;=\; \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2} \;=\; 1\,.$$

Unit vector inverse transform

Stan divides an unconstrained vector $y \in \mathbb{R}^n$ by its norm, $\|y\| = \sqrt{y^\top y}$, to obtain a unit vector x,

$$x = \frac{y}{\|y\|}.$$

To generate a unit vector, Stan generates points at random in $\mathbb{R}^n$ with independent unit normal distributions, which are then standardized by dividing by their Euclidean length. Muller (1959) showed this generates points uniformly at random on S^{n-1}. That is, if we draw $y_n \sim \mathsf{Normal}(0, 1)$ for $n \in 1{:}n$, then $x = \frac{y}{\|y\|}$ has a uniform distribution over S^{n-1}. This allows us to use an n-dimensional basis for S^{n-1} that preserves local neighborhoods in that points that are close to each other in $\mathbb{R}^n$ map to points near each other in S^{n-1}. The mapping is not perfectly distance preserving, because there are points arbitrarily far away from each other in $\mathbb{R}^n$ that map to identical points in S^{n-1}.

Warning: undefined at zero!
The above mapping from $\mathbb{R}^n$ to S^n is not defined at zero. While this point outcome has measure zero during sampling, and may thus be ignored, it is the default initialization point and thus unit vector parameters cannot be initialized at zero. A simple workaround is to initialize from a very small interval around zero, which is an option built into all of the Stan interfaces.

Absolute Jacobian determinant of the unit vector inverse transform
The Jacobian matrix relating the input vector y to the output vector x is singular because $x^\top x = 1$ for any non-zero input vector y. Thus, there technically is no unique transformation from x to y. To circumvent this issue, let $r = \sqrt{y^\top y}$ so that $y = rx$. The transformation from (r, x_{-n}) to y is well-defined but r is arbitrary, so we set $r = 1$. In this case, the determinant of the Jacobian is proportional to $e^{-\frac{1}{2}y^\top y}$, which is the kernel of a standard multivariate normal distribution with n independent dimensions.

10.10. Correlation matrices

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

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

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

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

for all $k \in \{1, \ldots, K\}$, and it must be positive definite, so that for every non-zero K-vector a,

$$a^\top x a > 0.$$

The number of free parameters required to specify a $K \times K$ correlation matrix is $\binom{K}{2}$.

There is more than one way to map from $\binom{K}{2}$ unconstrained parameters to a $K \times K$ correlation matrix. Stan implements the Lewandowski-Kurowicka-Joe (LKJ) transform Lewandowski, Kurowicka, and Joe (2009).

Correlation matrix inverse transform

It is easiest to specify the inverse, going from its $\binom{K}{2}$ parameter basis to a correlation matrix. The basis will actually be broken down into two steps. To start, suppose y is a vector containing $\binom{K}{2}$ unconstrained values. These are first transformed via the bijective function $\tanh : \mathbb{R} \to (-1, 1)$

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

Then, define a $K \times K$ matrix z, the upper triangular values of which are filled by row with the transformed values, and the diagonal entries are set to one. For example, in the 4×4 case, there are $\binom{4}{2}$ values arranged as

$$z = \begin{bmatrix} 1 & \tanh y_1 & \tanh y_2 & \tanh y_4 \\ 0 & 1 & \tanh y_3 & \tanh y_5 \\ 0 & 0 & 1 & \tanh y_6 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Lewandowski, Kurowicka and Joe (LKJ) show how to bijectively map the array z to a correlation matrix x. The entry $z_{i,j}$ for $i < j$ is interpreted as the canonical partial correlation (CPC) between i and j, which is the correlation between i's residuals and j's residuals when both i and j are regressed on all variables i' such that $i' < i$. In the case of $i = 1$, there are no earlier variables, so $z_{1,j}$ is just the Pearson correlation between i and j.

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

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

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

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

Given the upper-triangular Cholesky factor w, the final correlation matrix is

$$x = w^\top w.$$

Lewandowski, Kurowicka, and Joe (2009) 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

From the inverse of equation 11 in (Lewandowski, Kurowicka, and Joe 2009), the absolute Jacobian determinant is

$$\sqrt{\prod_{i=1}^{K-1} \prod_{j=i+1}^{K} \left(1 - z_{i,j}^2\right)^{K-i-1}} \times \prod_{i=1}^{K-1} \prod_{j=i+1}^{K} \frac{\partial z_{i,j}}{\partial y_{i,j}}$$

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 canonical partial correlations (CPCs) is simplified by the ordering of the elements in the definition of w, which when inverted yields

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

The final stage of the transform reverses the hyperbolic tangent transform, which is defined by

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

The inverse hyperbolic tangent function, $\tanh^{-1}$, is also called the Fisher transformation.

10.11. 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]

[2] An alternative to the transform in this section, which can be coded directly in Stan, is to parameterize a covariance matrix as a scaled correlation matrix. An arbitrary $K \times K$ covariance matrix Σ can be

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

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

---

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

$$\frac{\partial}{\partial z_{m,n}} \left(z\, z^{\top} \right)_{m,n} \;=\; \frac{\partial}{\partial z_{m,n}} \left(\sum_{k=1}^{K} z_{m,k}\, z_{n,k} \right) \;=\; \begin{cases} 2\, z_{n,n} & \text{if } m = n \text{ and} \\ z_{n,n} & \text{if } m > n. \end{cases}$$

The absolute Jacobian determinant of the second transform is thus

$$2^K \prod_{m=1}^{K} \prod_{n=1}^{m} z_{n,n} \;=\; \prod_{n=1}^{K} \prod_{m=n}^{K} z_{n,n} \;=\; 2^K \prod_{k=1}^{K} z_{k,k}^{K-k+1}.$$

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,

$$\left(\prod_{k=1}^{K} z_{k,k} \right) \left(2^K \prod_{k=1}^{K} z_{k,k}^{K-k+1} \right) \;=\; 2^K \prod_{k=1}^{K} z_{k,k}^{K-k+2}.$$

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

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

10.12. Cholesky factors of covariance matrices

An $M \times M$ covariance matrix Σ can be Cholesky factored to a lower triangular matrix L such that $L\,L^\top = \Sigma$. If Σ is positive definite, then L will be $M \times M$. If Σ is only positive semi-definite, then L will be $M \times N$, with $N < M$.

A matrix is a Cholesky factor for a covariance matrix if and only if it is lower triangular, the diagonal entries are positive, and $M \geq N$. A matrix satisfying these conditions ensures that $L\,L^\top$ is positive semi-definite if $M > N$ and positive definite if $M = N$.

A Cholesky factor of a covariance matrix requires $N + \binom{N}{2} + (M - N)N$ unconstrained parameters.

Cholesky factor of covariance matrix transform

Stan's Cholesky factor transform only requires the first step of the covariance matrix transform, namely log transforming the positive diagonal elements. Suppose x is an $M \times N$ Cholesky factor. The above-diagonal entries are zero, the diagonal entries are positive, and the below-diagonal entries are unconstrained. The transform required is thus

$$y_{m,n} = \begin{cases} 0 & \text{if } m < n, \\ \log x_{m,m} & \text{if } m = n, \text{ and} \\ x_{m,n} & \text{if } m > n. \end{cases}$$

Cholesky factor of covariance matrix inverse transform

The inverse transform need only invert the logarithm with an exponentiation. If y is the unconstrained matrix representation, then the elements of the constrained matrix x is defined by

$$x_{m,n} = \begin{cases} 0 & \text{if } m < n, \\ \exp(y_{m,m}) & \text{if } m = n, \text{ and} \\ y_{m,n} & \text{if } m > n. \end{cases}$$

Absolute Jacobian determinant of Cholesky factor inverse transform

The transform has a diagonal Jacobian matrix, the absolute determinant of which is

$$\prod_{n=1}^{N} \frac{\partial}{\partial_{y_{n,n}}} \exp(y_{n,n}) \;=\; \prod_{n=1}^{N} \exp(y_{n,n}) \;=\; \prod_{n=1}^{N} x_{n,n}.$$

Let $x = f^{-1}(y)$ be the inverse transform from a $N + \binom{N}{2} + (M - N)N$ vector to an $M \times N$ Cholesky factor for a covariance matrix x defined in the previous section. A density function $p_X(x)$ defined on $M \times N$ Cholesky factors of covariance matrices is transformed to the density $p_Y(y)$ over $N + \binom{N}{2} + (M - N)N$ vectors y by

$$p_Y(y) = p_X(f^{-1}(y)) \prod_{N=1}^{N} x_{n,n}.$$

10.13. Cholesky factors of correlation matrices

A $K \times K$ correlation matrix Ω is positive definite and has a unit diagonal. Because it is positive definite, it can be Cholesky factored to a $K \times K$ lower-triangular matrix L with positive diagonal elements such that $\Omega = L\,L^\top$. Because the correlation matrix has a unit diagonal,

$$\Omega_{k,k} = L_k\,L_k^\top = 1,$$

each row vector L_k of the Cholesky factor is of unit length. The length and positivity constraint allow the diagonal elements of L to be calculated from the off-diagonal elements, so that a Cholesky factor for a $K \times K$ correlation matrix requires only $\binom{K}{2}$ unconstrained parameters.

Cholesky factor of correlation matrix inverse transform

It is easiest to start with the inverse transform from the $\binom{K}{2}$ unconstrained parameters y to the $K \times K$ lower-triangular Cholesky factor x. The inverse transform is based on the hyperbolic tangent function, tanh, which satisfies $\tanh(x) \in (-1, 1)$. Here it will function like an inverse logit with a sign to pick out the direction of an underlying canonical partial correlation; see the section on correlation matrix transforms for more information on the relation between canonical partial correlations and the Cholesky factors of correlation matrices.

Suppose y is a vector of $\binom{K}{2}$ unconstrained values. Let z be a lower-triangular matrix with zero diagonal and below diagonal entries filled by row. For example, in the 3×3 case,

$$z = \begin{bmatrix} 0 & 0 & 0 \\ \tanh y_1 & 0 & 0 \\ \tanh y_2 & \tanh y_3 & 0 \end{bmatrix}$$

The matrix z, with entries in the range $(-1, 1)$, is then transformed to the Cholesky factor x, by taking[3]

$$x_{i,j} = \begin{cases} 0 & \text{if } i < j \quad \text{[above diagonal]} \\ \sqrt{1 - \sum_{j'<j} x_{i,j'}^2} & \text{if } i = j \quad \text{[on diagonal]} \\ z_{i,j} \sqrt{1 - \sum_{j'<j} x_{i,j'}^2} & \text{if } i > j \quad \text{[below diagonal]} \end{cases}$$

In the 3×3 case, this yields

$$x = \begin{bmatrix} 1 & 0 & 0 \\ z_{2,1} & \sqrt{1 - x_{2,1}^2} & 0 \\ z_{3,1} & z_{3,2}\sqrt{1 - x_{3,1}^2} & \sqrt{1 - (x_{3,1}^2 + x_{3,2}^2)} \end{bmatrix},$$

where the $z_{i,j} \in (-1, 1)$ are the tanh-transformed y.

The approach is a signed stick-breaking process on the quadratic (Euclidean length) scale. Starting from length 1 at $j = 1$, each below-diagonal entry $x_{i,j}$ is determined by the (signed) fraction $z_{i,j}$ of the remaining length for the row that it consumes. The diagonal entries $x_{i,i}$ get any leftover length from earlier entries in their row. The above-diagonal entries are zero.

Cholesky factor of correlation matrix transform

Suppose x is a $K \times K$ Cholesky factor for some correlation matrix. The first step of the transform reconstructs the intermediate values z from x,

$$z_{i,j} = \frac{x_{i,j}}{\sqrt{1 - \sum_{j'<j} x_{i,j'}^2}}.$$

The mapping from the resulting z to y inverts tanh,

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

[3]For convenience, a summation with no terms, such as $\sum_{j'<1} x_{i,j'}$, is defined to be 0. This implies $x_{1,1} = 1$ and that $x_{i,1} = z_{i,1}$ for $i > 1$.

Absolute Jacobian determinant of inverse transform

The Jacobian of the full transform is the product of the Jacobians of its component transforms.

First, for the inverse transform $z = \tanh y$, the derivative is

$$\frac{d}{dy} \tanh y = \frac{1}{(\cosh y)^2}.$$

Second, for the inverse transform of z to x, the resulting Jacobian matrix J is of dimension $\binom{K}{2} \times \binom{K}{2}$, with indexes (i, j) for $(i > j)$. The Jacobian matrix is lower triangular, so that its determinant is the product of its diagonal entries, of which there is one for each (i, j) pair,

$$|\det J| = \prod_{i>j} \left| \frac{d}{dz_{i,j}} x_{i,j} \right|,$$

where

$$\frac{d}{dz_{i,j}} x_{i,j} = \sqrt{1 - \sum_{j'<j} x_{i,j'}^2}.$$

So the combined density for unconstrained y is

$$p_Y(y) = p_X(f^{-1}(y)) \prod_{n<\binom{K}{2}} \frac{1}{(\cosh y)^2} \prod_{i>j} \left(1 - \sum_{j'<j} x_{i,j'}^2 \right)^{1/2},$$

where $x = f^{-1}(y)$ is used for notational convenience. The log Jacobian determinant of the complete inverse transform $x = f^{-1}(y)$ is given by

$$\log |\det J| = -2 \sum_{n \leq \binom{K}{2}} \log \cosh y \; + \; \frac{1}{2} \sum_{i>j} \log \left(1 - \sum_{j'<j} x_{i,j'}^2 \right).$$

11. 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.

11.1. BNF grammars

Syntactic conventions

In the following BNF grammars, tokens are represented in ALLCAPS. Grammar non-terminals are surrounded by < and >. A square brackets (`[A]`) indicates optionality of `A`. A postfixed Kleene star (`A*`) indicates zero or more occurrences of `A`. Parenthesis can be used to group symbols together in productions.

Finally, this grammar uses the concept of "parameterized nonterminals" as used in the parsing library Menhir. A rule like `<list(x)> ::= x (COMMA x)*` declares a generic list rule, which can later be applied to others by the symbol `<list(<expression>)>`.

The following representation is constructed directly from the OCaml reference parser using a tool called Obelisk. The raw output is available here.

Programs

```
<program> ::= [<function_block>] [<data_block>] [<transformed_data_block>]
              [<parameters_block>] [<transformed_parameters_block>]
              [<model_block>] [<generated_quantities_block>] EOF

<functions_only> ::= <function_def>* EOF

<function_block> ::= FUNCTIONBLOCK LBRACE <function_def>* RBRACE

<data_block> ::= DATABLOCK LBRACE <top_var_decl_no_assign>* RBRACE

<transformed_data_block> ::= TRANSFORMEDDATABLOCK LBRACE
                             <top_vardecl_or_statement>* RBRACE

<parameters_block> ::= PARAMETERSBLOCK LBRACE <top_var_decl_no_assign>*
                       RBRACE
```

```
<transformed_parameters_block> ::= TRANSFORMEDPARAMETERSBLOCK LBRACE
                                   <top_vardecl_or_statement>* RBRACE

<model_block> ::= MODELBLOCK LBRACE <vardecl_or_statement>* RBRACE

<generated_quantities_block> ::= GENERATEDQUANTITIESBLOCK LBRACE
                                 <top_vardecl_or_statement>* RBRACE
```

Function declarations and definitions

```
<function_def> ::= <return_type> <decl_identifier> LPAREN [<arg_decl> (COMMA
                   <arg_decl>)*] RPAREN <statement>

<return_type> ::= VOID
                | <unsized_type>

<arg_decl> ::= [DATABLOCK] <unsized_type> <decl_identifier>

<unsized_type> ::= ARRAY <unsized_dims> <basic_type>
                 | ARRAY <unsized_dims> <unsized_tuple_type>
                 | <basic_type>
                 | <unsized_tuple_type>

<unsized_tuple_type> ::= TUPLE LPAREN <unsized_type> COMMA <unsized_type>
                         (COMMA <unsized_type>)* RPAREN

<basic_type> ::= INT
               | REAL
               | COMPLEX
               | VECTOR
               | ROWVECTOR
               | MATRIX
               | COMPLEXVECTOR
               | COMPLEXROWVECTOR
               | COMPLEXMATRIX

<unsized_dims> ::= LBRACK COMMA* RBRACK
```

Variable declarations and compound definitions

```
<identifier> ::= IDENTIFIER
               | TRUNCATE

<decl_identifier> ::= <identifier>
```

```
<no_assign> ::= UNREACHABLE

<optional_assignment(rhs)> ::= [ASSIGN rhs]

<id_and_optional_assignment(rhs)> ::= <decl_identifier>
                                      <optional_assignment(rhs)>

<decl(type_rule, rhs)> ::= type_rule <decl_identifier> <dims>
                           <optional_assignment(rhs)> SEMICOLON
                         | <higher_type(type_rule)>
                           <id_and_optional_assignment(rhs)> (COMMA
                           <id_and_optional_assignment(rhs)>)* SEMICOLON

<higher_type(type_rule)> ::= <array_type(type_rule)>
                           | <tuple_type(type_rule)>
                           | type_rule

<array_type(type_rule)> ::= <arr_dims> type_rule
                          | <arr_dims> <tuple_type(type_rule)>

<tuple_type(type_rule)> ::= TUPLE LPAREN <higher_type(type_rule)> COMMA
                            <higher_type(type_rule)> (COMMA
                            <higher_type(type_rule)>)* RPAREN

<var_decl> ::= <decl(<sized_basic_type>, <expression>)>

<top_var_decl> ::= <decl(<top_var_type>, <expression>)>

<top_var_decl_no_assign> ::= <decl(<top_var_type>, <no_assign>)>
                           | SEMICOLON

<sized_basic_type> ::= INT
                     | REAL
                     | COMPLEX
                     | VECTOR LBRACK <expression> RBRACK
                     | ROWVECTOR LBRACK <expression> RBRACK
                     | MATRIX LBRACK <expression> COMMA <expression> RBRACK
                     | COMPLEXVECTOR LBRACK <expression> RBRACK
                     | COMPLEXROWVECTOR LBRACK <expression> RBRACK
                     | COMPLEXMATRIX LBRACK <expression> COMMA <expression>
                       RBRACK

<top_var_type> ::= INT [LABRACK <range> RABRACK]
```

```
                | REAL <type_constraint>                  | TUPLE

                | COMPLEX <type_constraint>
                | VECTOR <type_constraint> LBRACK <expression> RBRACK
                | ROWVECTOR <type_constraint> LBRACK <expression> RBRACK
                | MATRIX <type_constraint> LBRACK <expression> COMMA
                  <expression> RBRACK
                | COMPLEXVECTOR <type_constraint> LBRACK <expression> RBRACK
                | COMPLEXROWVECTOR <type_constraint> LBRACK <expression>
                  RBRACK
                | COMPLEXMATRIX <type_constraint> LBRACK <expression> COMMA
                  <expression> RBRACK
                | ORDERED LBRACK <expression> RBRACK
                | POSITIVEORDERED LBRACK <expression> RBRACK
                | SIMPLEX LBRACK <expression> RBRACK
                | UNITVECTOR LBRACK <expression> RBRACK
                | CHOLESKYFACTORCORR LBRACK <expression> RBRACK
                | CHOLESKYFACTORCOV LBRACK <expression> [COMMA <expression>]
                  RBRACK
                | CORRMATRIX LBRACK <expression> RBRACK
                | COVMATRIX LBRACK <expression> RBRACK

<type_constraint> ::= [LABRACK <range> RABRACK]
                    | LABRACK <offset_mult> RABRACK

<range> ::= LOWER ASSIGN <constr_expression> COMMA UPPER ASSIGN
            <constr_expression>
          | UPPER ASSIGN <constr_expression> COMMA LOWER ASSIGN
            <constr_expression>
          | LOWER ASSIGN <constr_expression>
          | UPPER ASSIGN <constr_expression>

<offset_mult> ::= OFFSET ASSIGN <constr_expression> COMMA MULTIPLIER ASSIGN
                  <constr_expression>
                | MULTIPLIER ASSIGN <constr_expression> COMMA OFFSET ASSIGN
                  <constr_expression>
                | OFFSET ASSIGN <constr_expression>
                | MULTIPLIER ASSIGN <constr_expression>

<arr_dims> ::= ARRAY LBRACK <expression> (COMMA <expression>)* RBRACK
```

Expressions

```
<expression> ::= <expression> QMARK <expression> COLON <expression>
```

```
                | <expression> <infixOp> <expression>
                | <prefixOp> <expression>
                | <expression> <postfixOp>
                | <common_expression>

<constr_expression> ::= <constr_expression> <arithmeticBinOp>
                          <constr_expression>
                        | <prefixOp> <constr_expression>
                        | <constr_expression> <postfixOp>
                        | <common_expression>

<common_expression> ::= <identifier>
                        | INTNUMERAL
                        | REALNUMERAL
                        | DOTNUMERAL
                        | IMAGNUMERAL
                        | LBRACE <expression> (COMMA <expression>)* RBRACE
                        | LBRACK [<expression> (COMMA <expression>)*] RBRACK
                        | <identifier> LPAREN [<expression> (COMMA
                          <expression>)*] RPAREN
                        | TARGET LPAREN RPAREN
                        | <identifier> LPAREN <expression> BAR [<expression>
                          (COMMA <expression>)*] RPAREN
                        | LPAREN <expression> COMMA <expression> (COMMA
                          <expression>)* RPAREN
                        | <common_expression> DOTNUMERAL
                        | <common_expression> LBRACK <indexes> RBRACK
                        | LPAREN <expression> RPAREN

<prefixOp> ::= BANG
             | MINUS
             | PLUS

<postfixOp> ::= TRANSPOSE

<infixOp> ::= <arithmeticBinOp>
            | <logicalBinOp>

<arithmeticBinOp> ::= PLUS
                    | MINUS
                    | TIMES
                    | DIVIDE
                    | IDIVIDE
```

```
                    | MODULO
                    | LDIVIDE
                    | ELTTIMES
                    | ELTDIVIDE
                    | HAT
                    | ELTPOW

<logicalBinOp> ::= OR
                 | AND
                 | EQUALS
                 | NEQUALS
                 | LABRACK
                 | LEQ
                 | RABRACK
                 | GEQ

<indexes> ::= epsilon
            | COLON
            | <expression>
            | <expression> COLON
            | COLON <expression>
            | <expression> COLON <expression>
            | <indexes> COMMA <indexes>

<printables> ::= <expression>
               | <string_literal>
               | <printables> COMMA <printables>
```

Statements

```
<statement> ::= <atomic_statement>
              | <nested_statement>

<atomic_statement> ::= <common_expression> <assignment_op> <expression>
                       SEMICOLON
                     | <identifier> LPAREN [<expression> (COMMA
                       <expression>)*] RPAREN SEMICOLON
                     | <expression> TILDE <identifier> LPAREN [<expression>
                       (COMMA <expression>)*] RPAREN [<truncation>] SEMICOLON
                     | TARGET PLUSASSIGN <expression> SEMICOLON
                     | BREAK SEMICOLON
                     | CONTINUE SEMICOLON
                     | PRINT LPAREN <printables> RPAREN SEMICOLON
                     | REJECT LPAREN <printables> RPAREN SEMICOLON
```

```
                    | FATAL_ERROR LPAREN <printables> RPAREN SEMICOLON
                    | RETURN <expression> SEMICOLON
                    | RETURN SEMICOLON
                    | SEMICOLON

<assignment_op> ::= ASSIGN
                  | PLUSASSIGN
                  | MINUSASSIGN
                  | TIMESASSIGN
                  | DIVIDEASSIGN
                  | ELTTIMESASSIGN
                  | ELTDIVIDEASSIGN

<string_literal> ::= STRINGLITERAL

<truncation> ::= TRUNCATE LBRACK [<expression>] COMMA [<expression>] RBRACK

<nested_statement> ::= IF LPAREN <expression> RPAREN <vardecl_or_statement>
                       ELSE <vardecl_or_statement>
                     | IF LPAREN <expression> RPAREN <vardecl_or_statement>
                     | WHILE LPAREN <expression> RPAREN
                       <vardecl_or_statement>
                     | FOR LPAREN <identifier> IN <expression> COLON
                       <expression> RPAREN <vardecl_or_statement>
                     | FOR LPAREN <identifier> IN <expression> RPAREN
                       <vardecl_or_statement>
                     | PROFILE LPAREN <string_literal> RPAREN LBRACE
                       <vardecl_or_statement>* RBRACE
                     | LBRACE <vardecl_or_statement>* RBRACE

<vardecl_or_statement> ::= <statement>
                         | <var_decl>

<top_vardecl_or_statement> ::= <statement>
                             | <top_var_decl>
```

11.2. Tokenizing rules

Many of the tokens used in the BNF grammars follow obviously from their names: `DATABLOCK` is the literal string 'data', `COMMA` is a single ',' character, etc. The literal representation of each operator is additionally provided in the operator precedence table.

A few tokens are not so obvious, and are defined here in regular expressions:

```
IDENTIFIER = [a-zA-Z] [a-zA-Z0-9_]*

STRINGLITERAL = ".*"

INTNUMERAL = [0-9]+ (_ [0-9]+)*

EXPLITERAL = [eE] [+-]? INTNUMERAL

REALNUMERAL = INTNUMERAL \. INTNUMERAL? EXPLITERAL?
            | \. INTNUMERAL EXPLITERAL
            | INTNUMERAL EXPLITERAL

IMAGNUMERAL = (REALNUMERAL | INTNUMERAL) i

DOTNUMERAL = \. INTNUMERAL
```

11.3. 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 compilation with an error message indicating the location of the problem.

Operator precedence and associativity

In the Stan grammar provided in this chapter, the expression `1 + 2 * 3` has two parses. As described in the operator precedence table, Stan disambiguates between the meaning $1 + (2 \times 3)$ and the meaning $(1 + 2) \times 3$ based on operator precedences and associativities.

Typing of compound declaration and definition

In a compound variable declaration and definition, the type of the right-hand side expression must be assignable to the variable being declared. The assignability constraint restricts compound declarations and definitions to local variables and variables declared in the transformed data, transformed parameters, and generated quantities blocks.

Typing of array expressions

The types of expressions used for elements in array expressions (`'{' expressions '}'`) must all be of the same type or a mixture of scalar (`int`, `real` and `complex`) types (in which case the result is promoted to be of the highest type on the `int -> real -> complex` hierarchy).

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.

For loop containers

The for loop statement requires that we specify in addition to the loop identifier, either a range consisting of two expressions denoting an integer, separated by ':', or a single expression denoting a container. The loop variable will be of type integer in the former case and of the contained type in the latter case. Furthermore, the loop variable must not be in scope (i.e., there is no masking of variables).

Print arguments

The arguments to a print statement cannot be void.

Only break and continue in loops

The `break` and `continue` statements may only be used within the body of a for-loop or while-loop.

Block-specific restrictions

Some constructs in the Stan language are only allowed in certain blocks or in certain kinds of user-defined functions.

PRNG functions

Functions ending in `_rng` may only be called in the `transformed data` and `generated quantities` block, and within the bodies of user-defined functions with names ending in `_rng`.

Unnormalized distributions

Unnormalized distributions (with suffixes `_lupmf` or `_lupdf`) may only be called in the `model` block, user-defined probability functions, or within the bodies of user defined functions which end in `_lp`.

Incrementing and accessing target

`target +=` statements can only be used inside of the `model` block or user-defined functions which end in `_lp`.

User defined functions which end in `_lp` and the `target()` function can only be used in the `model` block, `transformed parameters` block, and in the bodies of other user defined functions which end in `_lp`.

Sampling statements (using ~) can only be used in the `model` block or in the bodies of user-defined functions which end in `_lp`.

Probability function naming

A probability function literal must have one of the following suffixes: `_lpdf`, `_lpmf`, `_lcdf`, or `_lccdf`.

Indexes

Standalone expressions used as indexes must denote either an integer (`int`) or an integer array (`array[] int`). Expressions participating in range indexes (e.g., `a` and `b` in `a : b`) must denote integers (`int`).

A second condition is that there not be more indexes provided than dimensions of the underlying expression (in general) or variable (on the left side of assignments) being indexed. A vector or row vector adds 1 to the array dimension and a matrix adds 2. That is, the type `array[ , , ] matrix`, a three-dimensional array of matrices, has five index positions: three for the array, one for the row of the matrix and one for the column.

12. Program Execution

This chapter provides a sketch of how a compiled Stan model is executed using sampling. Optimization shares the same data reading and initialization steps, but then does optimization rather than sampling.

This sketch is elaborated in the following chapters of this part, which cover variable declarations, expressions, statements, and blocks in more detail.

12.1. Reading and transforming data

The reading and transforming data steps are the same for sampling, optimization and diagnostics.

Read data

The first step of execution is to read data into memory. Data may be read in through file (in CmdStan) or through memory (RStan and PyStan); see their respective manuals for details.[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.

Transformed data variables are initialized with real values set to `NaN` and integer values set to the smallest integer (large absolute value negative number).

[1] The C++ code underlying Stan is flexible enough to allow data to be read from memory or file. Calls from R, for instance, can be configured to read data from file or directly from R's memory.

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.

12.2. Initialization

Initialization is the same for all of Stan's algorithms.

User-supplied initial values

If there are user-supplied initial values for parameters, these are read using the same input mechanism and same file format as data reads. Any constraints declared on the parameters are validated for the initial values. If a variable's value violates its declared constraint, the program halts and a diagnostic message is printed.

After being read, initial values are transformed to unconstrained values that will be used to initialize the sampler.

Boundary values are problematic

Because of the way Stan defines its transforms from the constrained to the unconstrained space, initializing parameters on the boundaries of their constraints is usually problematic. For instance, with a constraint

```
parameters {
  real<lower=0, upper=1> theta;
  // ...
}
```

an initial value of 0 for `theta` leads to an unconstrained value of $-\infty$, whereas a value of 1 leads to an unconstrained value of $+\infty$. While this will be inverse transformed back correctly given the behavior of floating point arithmetic, the Jacobian will be infinite and the log probability function will fail and raise an exception.

Random initial values

If there are no user-supplied initial values, the default initialization strategy is to initialize the unconstrained parameters directly with values drawn uniformly from the interval $(-2, 2)$. The bounds of this initialization can be changed but it is always symmetric around 0. The value of 0 is special in that it represents the median of the initialization. An unconstrained value of 0 corresponds to different parameter values depending on the constraints declared on the parameters.

An unconstrained real does not involve any transform, so an initial value of 0 for the unconstrained parameters is also a value of 0 for the constrained parameters.

For parameters that are bounded below at 0, the initial value of 0 on the unconstrained scale corresponds to $\exp(0) = 1$ on the constrained scale. A value of -2 corresponds to $\exp(-2) = .13$ and a value of 2 corresponds to $\exp(2) = 7.4$.

For parameters bounded above and below, the initial value of 0 on the unconstrained scale corresponds to a value at the midpoint of the constraint interval. For probability parameters, bounded below by 0 and above by 1, the transform is the inverse logit, so that an initial unconstrained value of 0 corresponds to a constrained value of 0.5, -2 corresponds to 0.12 and 2 to 0.88. Bounds other than 0 and 1 are just scaled and translated.

Simplexes with initial values of 0 on the unconstrained basis correspond to symmetric values on the constrained values (i.e., each value is $1/K$ in a K-simplex).

Cholesky factors for positive-definite matrices are initialized to 1 on the diagonal and 0 elsewhere; this is because the diagonal is log transformed and the below-diagonal values are unconstrained.

The initial values for other parameters can be determined from the transform that is applied. The transforms are all described in full detail in the chapter on variable transforms.

Zero initial values

The initial values may all be set to 0 on the unconstrained scale. This can be helpful for diagnosis, and may also be a good starting point for sampling. Once a model is running, multiple chains with more diffuse starting points can help diagnose problems with convergence; see the user's guide for more information on convergence monitoring.

12.3. Sampling

Sampling is based on simulating the Hamiltonian of a particle with a starting position equal to the current parameter values and an initial momentum (kinetic energy) generated randomly. The potential energy at work on the particle is taken to be the negative log (unnormalized) total probability function defined by the model. In the usual approach to implementing HMC, the Hamiltonian dynamics of the particle is simulated using the leapfrog integrator, which discretizes the smooth path of the particle into a number of small time steps called leapfrog steps.

Leapfrog steps

For each leapfrog step, the negative log probability function and its gradient need to be evaluated at the position corresponding to the current parameter values (a more detailed sketch is provided in the next section). These are used to update the

momentum based on the gradient and the position based on the momentum.

For simple models, only a few leapfrog steps with large step sizes are needed. For models with complex posterior geometries, many small leapfrog steps may be needed to accurately model the path of the parameters.

If the user specifies the number of leapfrog steps (i.e., chooses to use standard HMC), that number of leapfrog steps are simulated. If the user has not specified the number of leapfrog steps, the No-U-Turn sampler (NUTS) will determine the number of leapfrog steps adaptively (Hoffman and Gelman 2014).

Log probability and gradient calculation

During each leapfrog step, the log probability function and its gradient must be calculated. This is where most of the time in the Stan algorithm is spent. This log probability function, which is used by the sampling algorithm, is defined over the unconstrained parameters.

The first step of the calculation requires the inverse transform of the unconstrained parameter values back to the constrained parameters in terms of which the model is defined. There is no error checking required because the inverse transform is a total function on every point in whose range satisfies the constraints.

Because the probability statements in the model are defined in terms of constrained parameters, the log Jacobian of the inverse transform must be added to the accumulated log probability.

Next, the transformed parameter statements are executed. After they complete, any constraints declared for the transformed parameters are checked. If the constraints are violated, the model will halt with a diagnostic error message.

The final step in the log probability function calculation is to execute the statements defined in the model block.

As the log probability function executes, it accumulates an in-memory representation of the expression tree used to calculate the log probability. This includes all of the transformed parameter operations and all of the Jacobian adjustments. This tree is then used to evaluate the gradients by propagating partial derivatives backward along the expression graph. The gradient calculations account for the majority of the cycles consumed by a Stan program.

Metropolis accept/reject

A standard Metropolis accept/reject step is required to retain detailed balance and ensure draws 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.

12.4. Optimization

Optimization runs very much like sampling in that it starts by reading the data and then initializing parameters. Unlike sampling, it produces a deterministic output which requires no further analysis other than to verify that the optimizer itself converged to a posterior mode. The output for optimization is also similar to that for sampling.

12.5. Variational inference

Variational inference also runs similar to sampling. It begins by reading the data and initializing the algorithm. The initial variational approximation is a random draw from the standard normal distribution in the unconstrained (real-coordinate) space. Again, similar to sampling, it outputs draws from the approximate posterior once the algorithm has decided that it has converged. Thus, the tools we use for analyzing the result of Stan's sampling routines can also be used for variational inference.

12.6. Model diagnostics

Model diagnostics are like sampling and optimization in that they depend on a model's data being read and its parameters being initialized. The user's guides for the interfaces (RStan, PyStan, CmdStan) provide more details on the diagnostics available; as of Stan 2.0, that's just gradients on the unconstrained scale and log probabilities.

12.7. Output

For each final draw (not counting draws during warmup or draws that are thinned), there is an output stage of writing the draw.

Generated quantities

Before generating any output, the statements in the generated quantities block are executed. This can be used for any forward simulation based on parameters of the model. Or it may be used to transform parameters to an appropriate form for output.

After the generated quantities statements execute, the constraints declared on generated quantities variables are validated. If these constraints are violated, the program will terminate with a diagnostic message.

Write

The final step is to write the actual values. The values of all variables declared as parameters, transformed parameters, or generated quantities are written. Local variables are not written, nor is the data or transformed data. All values are written in their constrained forms, that is the form that is used in the model definitions.

In the executable form of a Stan models, parameters, transformed parameters, and generated quantities are written to a file in comma-separated value (CSV) notation with a header defining the names of the parameters (including indices for multivariate parameters).[2]

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

13. Deprecated Features

This appendix lists currently deprecated functionality along with how to replace it.

Starting with Stan 2.29, minor (syntax-level) deprecations can be removed 3 versions after release; e.g., syntax deprecated in Stan 2.20 will be removed in Stan 2.23 and placed in Removed Features. The Stan compiler can automatically update many of these on the behalf of the user for at least one version after they are removed.

Any feature which changes semantic meaning (such as the upgraded ODE solver interface) will not be removed until a major version change (e.g., Stan 3.0).

13.1. `lkj_cov` distribution

Deprecated:The distribution `lkj_cov` is deprecated.

Replacement: Replace `lkj_cov_lpdf(...)` with an `lkj_corr` distribution on the correlation matrix and independent lognormal distributions on the scales. That is, replace

```
cov_matrix[K] Sigma;
// ...
Sigma ~ lkj_cov(mu, tau, eta);
```

with

```
corr_matrix[K] Omega;
vector<lower=0>[K] sigma;
// ...
Omega ~ lkj_corr(eta);
sigma ~ lognormal(mu, tau);
// ...
cov_matrix[K] Sigma;
Sigma <- quad_form_diag(Omega, sigma);
```

The variable `Sigma` may be defined as a local variable in the model block or as a transformed parameter. An even more efficient transform would use Cholesky factors rather than full correlation matrix types.

Scheduled Removal: Stan 3.0 or later.

13.2. New Keywords

Deprecated: The following identifiers will become reserved in the language in the specified version.

Replacement: Rename any variables or functions with these names.

Identifier	Version
jacobian	2.38

13.3. Deprecated Functions

Several built-in Stan functions have been deprecated. Consult the functions reference for more information.

14. Removed Features

This chapter lists functionalities that were once present in the language but have since been removed, along with how to replace them.

14.1. `lp__` variable

Removed: The variable `lp__` is no longer available for direct access or manipulation.

Replacement: General manipulation of the value of the `lp__` variable is not allowed, but

```
lp__ <- lp__ + e;
```

can be replaced with

```
target += e;
```

The value of `lp__` is available through the no-argument function `target()`.

14.2. Assignment with `<-`

Removed: The operator `<-` for assignment, e.g.,

```
a <- b;
```

is no longer available.

Replacement: The new syntax uses the operator `=` for assignment, e.g.,

```
a = b;
```

Removed In: Stan 2.33

14.3. `increment_log_prob` statement

Removed: The `increment_log_prob(u)` statement for incrementing the log density accumulator by `u` is no longer available.

Replacement: Replace the above statement with

```
target += u;
```

Removed In: Stan 2.33

14.4. `get_lp()` function

Removed: The built-in no-argument function `get_lp()` is no longer available.

Replacement: Use the no-argument function `target()` instead.

Removed In: Stan 2.33

14.5. `_log` density and mass functions

Removed: Formerly, the probability function for the distribution `foo` would be applied to an outcome variable `y` and sequence of zero or more parameters `...` to produce the expression `foo_log(y, ...)`. This suffix is no longer a special value.

Replacement: If `y` can be a real value (including vectors or matrices), replace

```
foo_log(y, ...)
```

with the log probability density function notation

```
foo_lpdf(y | ...).
```

If `y` must be an integer (including arrays), instead replace

```
foo_log(y, ...
```

with the log probability mass function

```
foo_lpmf(y | ...).
```

Removed In: Stan 2.33

14.6. `cdf_log` and `ccdf_log` cumulative distribution functions

Removed: The log cumulative distribution and complementary cumulative distribution functions for a distribution `foo` were formerly written as `foo_cdf_log` and `foo_ccdf_log`.

Replacement:

Replace `foo_cdf_log(y, ...)` with `foo_lcdf(y | ...)`.

Replace `foo_ccdf_log(y, ...)` with `foo_lccdf(y | ...)`.

14.7. User-defined function with `_log` suffix

Removed: A user-defined function ending in `_log` can be no longer be used in statements.qmd#distribution-statements.section.

Replacement: Replace the `_log` suffix with `_lpdf` for density functions or `_lpmf` for mass functions in the user-defined function.

Removed In: Stan 2.33

Note: Following Stan 2.33, users can stil define a function ending in `_log`, it simply no longer has a special meaning or is supported in the ~ syntax.

14.8. `if_else` function

Removed: The function `if_else` is no longer available.

Replacement: Use the conditional operator which allows more flexibility in the types of `b` and `c` and is much more efficient in that it only evaluates whichever of `b` or `c` is returned.

```
x = if_else(a, b, c);
```

with

```
x = a ? b : c;
```

Removed In: Stan 2.33

14.9. Character # as comment prefix

Removed: The use of `#` for line-based comments is no longer permitted. `#` may only be used for `#include` statements.

Replacement: Use a pair of forward slashes, `//`, for line comments.

Removed In: Stan 2.33

14.10. Postfix brackets array syntax

Before Stan 2.26, arrays were declared by writing syntax after the variable.

Removed: The use of array declarations like

```
int n[5];
real a[3, 4];
real<lower=0> z[5, 4, 2];
vector[7] mu[3];
matrix[7, 2] mu[15, 12];
cholesky_factor_cov[5, 6] mu[2, 3, 4];
```

Replacement: The use of the `array` keyword, which replaces the above examples

with

```
array[5] int n;
array[3, 4] real a;
array[5, 4, 2] real<lower=0> z;
array[3] vector[7] mu;
array[15, 12] matrix[7, 2] mu;
array[2, 3, 4] cholesky_factor_cov[5, 6] mu;
```

Removed In: Stan 2.33

14.11. Nested multiple indexing in assignments

Stan interprets nested indexing in assingments as flat indexing so that a statement like

```
a[:][1] = b;
```

is the same as

```
a[:,1] = b;
```

However, this is inconsistent with multiple indexing rules.

To avoid confusion nested multiple indexing in assignment became an error in Stan 2.33. Nesting single indexing is still allowed as it cannot lead to ambiguity.

Removed In: Stan 2.33

14.12. Real values in conditionals

Removed: Using a real value in a conditional is no longer permitted.

```
real x = 1.0;
if (x) {
```

The value was interpreted as true if it is nonzero.

Replacement: For the exact equivalent, use a comparison operator to make the intent clear.

```
real x = 1.0;
if (x != 0) {
```

However, one should keep in mind that floating point calculations are subject to rounding errors and precise equality is fragile. It is worth considering whether the

more robust alternative `abs(x) < machine_precision()` is appropriate for the use case.

Removed In: Stan 2.34

Part II

Algorithms

15. MCMC Sampling

This chapter presents the two Markov chain Monte Carlo (MCMC) algorithms used in Stan, the Hamiltonian Monte Carlo (HMC) algorithm and its adaptive variant the no-U-turn sampler (NUTS), along with details of their implementation and configuration.

15.1. Hamiltonian Monte Carlo

Hamiltonian Monte Carlo (HMC) is a Markov chain Monte Carlo (MCMC) method that uses the derivatives of the density function being sampled to generate efficient transitions spanning the posterior (see, e.g., Betancourt and Girolami (2013), Neal (2011) for more details). It uses an approximate Hamiltonian dynamics simulation based on numerical integration which is then corrected by performing a Metropolis acceptance step.

This section translates the presentation of HMC by Betancourt and Girolami (2013) into the notation of Gelman et al. (2013).

Target density

The goal of sampling is to draw from a density $p(\theta)$ for parameters θ. This is typically a Bayesian posterior $p(\theta|y)$ given data y, and in particular, a Bayesian posterior coded as a Stan program.

Auxiliary momentum variable

HMC introduces auxiliary momentum variables ρ and draws from a joint density

$$p(\rho, \theta) = p(\rho|\theta)p(\theta).$$

In most applications of HMC, including Stan, the auxiliary density is a multivariate normal that does not depend on the parameters θ,

$$\rho \sim \mathsf{MultiNormal}(0, M).$$

M is the Euclidean metric. It can be seen as a transform of parameter space that makes sampling more efficient; see Betancourt (2017) for details.

By default Stan sets M^{-1} equal to a diagonal estimate of the covariance computed during warmup.

The Hamiltonian

The joint density $p(\rho, \theta)$ defines a Hamiltonian

$$\begin{aligned} H(\rho, \theta) &= -\log p(\rho, \theta) \\ &= -\log p(\rho|\theta) - \log p(\theta). \\ &= T(\rho|\theta) + V(\theta), \end{aligned}$$

where the term

$$T(\rho|\theta) = -\log p(\rho|\theta)$$

is called the "kinetic energy" and the term

$$V(\theta) = -\log p(\theta)$$

is called the "potential energy." The potential energy is specified by the Stan program through its definition of a log density.

Generating transitions

Starting from the current value of the parameters θ, a transition to a new state is generated in two stages before being subjected to a Metropolis accept step.

First, a value for the momentum is drawn independently of the current parameter values,

$$\rho \sim \mathsf{MultiNormal}(0, M).$$

Thus momentum does not persist across iterations.

Next, the joint system (θ, ρ) made up of the current parameter values θ and new momentum ρ is evolved via Hamilton's equations,

$$\begin{aligned} \frac{d\theta}{dt} &= +\frac{\partial H}{\partial \rho} = +\frac{\partial T}{\partial \rho} \\ \frac{d\rho}{dt} &= -\frac{\partial H}{\partial \theta} = -\frac{\partial T}{\partial \theta} - \frac{\partial V}{\partial \theta}. \end{aligned}$$

With the momentum density being independent of the target density, i.e., $p(\rho|\theta) = p(\rho)$, the first term in the momentum time derivative, $\partial T/\partial\theta$ is zero, yielding the pair time derivatives

$$\begin{aligned} \frac{d\theta}{dt} &= +\frac{\partial T}{\partial \rho} \\ \frac{d\rho}{dt} &= -\frac{\partial V}{\partial \theta}. \end{aligned}$$

Leapfrog integrator

The last section leaves a two-state differential equation to solve. Stan, like most other HMC implementations, uses the leapfrog integrator, which is a numerical integration algorithm that's specifically adapted to provide stable results for Hamiltonian systems of equations.

Like most numerical integrators, the leapfrog algorithm takes discrete steps of some small time interval ϵ. The leapfrog algorithm begins by drawing a fresh momentum term independently of the parameter values θ or previous momentum value.

$$\rho \sim \mathsf{MultiNormal}(0, M).$$

It then alternates half-step updates of the momentum and full-step updates of the position.

$$\begin{aligned} \rho &\leftarrow \rho - \tfrac{\epsilon}{2}\tfrac{\partial V}{\partial \theta} \\ \theta &\leftarrow \theta + \epsilon\, M^{-1}\rho \\ \rho &\leftarrow \rho - \tfrac{\epsilon}{2}\tfrac{\partial V}{\partial \theta}. \end{aligned}$$

By applying L leapfrog steps, a total of $L\,\epsilon$ time is simulated. The resulting state at the end of the simulation (L repetitions of the above three steps) will be denoted (ρ^*, θ^*).

The leapfrog integrator's error is on the order of ϵ^3 per step and ϵ^2 globally, where ϵ is the time interval (also known as the step size); Leimkuhler and Reich (2004) provide a detailed analysis of numerical integration for Hamiltonian systems, including a derivation of the error bound for the leapfrog integrator.

Metropolis accept step

If the leapfrog integrator were perfect numerically, there would no need to do any more randomization per transition than generating a random momentum vector. Instead, what is done in practice to account for numerical errors during integration

is to apply a Metropolis acceptance step, where the probability of keeping the proposal (ρ^*, θ^*) generated by transitioning from (ρ, θ) is

$$\min(1,\, \exp(H(\rho, \theta) - H(\rho^*, \theta^*)))\,.$$

If the proposal is not accepted, the previous parameter value is returned for the next draw and used to initialize the next iteration.

Algorithm summary

The Hamiltonian Monte Carlo algorithm starts at a specified initial set of parameters θ; in Stan, this value is either user-specified or generated randomly. Then, for a given number of iterations, a new momentum vector is sampled and the current value of the parameter θ is updated using the leapfrog integrator with discretization time ϵ and number of steps L according to the Hamiltonian dynamics. Then a Metropolis acceptance step is applied, and a decision is made whether to update to the new state (θ^*, ρ^*) or keep the existing state.

15.2. HMC algorithm parameters

The Hamiltonian Monte Carlo algorithm has three parameters which must be set,

- discretization time ϵ,
- metric M, and
- number of steps taken L.

In practice, sampling efficiency, both in terms of iteration speed and iterations per effective sample, is highly sensitive to these three tuning parameters Neal (2011), Hoffman and Gelman (2014).

If ϵ is too large, the leapfrog integrator will be inaccurate and too many proposals will be rejected. If ϵ is too small, too many small steps will be taken by the leapfrog integrator leading to long simulation times per interval. Thus the goal is to balance the acceptance rate between these extremes.

If L is too small, the trajectory traced out in each iteration will be too short and sampling will devolve to a random walk. If L is too large, the algorithm will do too much work on each iteration.

If the inverse metric M^{-1} is a poor estimate of the posterior covariance, the step size ϵ must be kept small to maintain arithmetic precision. This would lead to a large L to compensate.

Integration time

The actual integration time is $L\,\epsilon$, a function of number of steps. Some interfaces to Stan set an approximate integration time t and the discretization interval (step size) ϵ. In these cases, the number of steps will be rounded down as

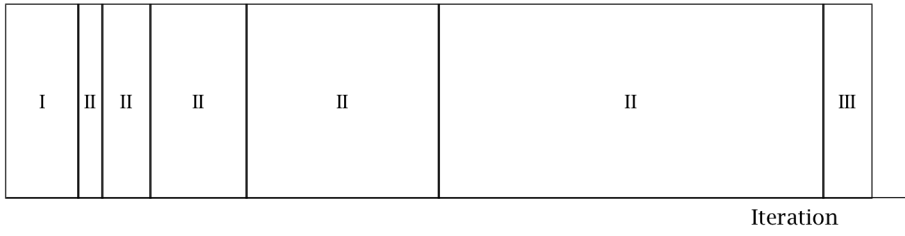
$$L = \left\lfloor \frac{t}{\epsilon} \right\rfloor .$$

and the actual integration time will still be $L\,\epsilon$.

Automatic parameter tuning

Stan is able to automatically optimize ϵ to match an acceptance-rate target, able to estimate M based on warmup sample iterations, and able to dynamically adapt L on the fly during sampling (and during warmup) using the no-U-turn sampling (NUTS) algorithm Hoffman and Gelman (2014).

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



When adaptation is engaged (it may be turned off by fixing a step size and metric), the warmup period is split into three stages, as illustrated in the warmup adaptation figure, with two *fast* intervals surrounding a series of growing *slow* intervals. Here fast and slow refer to parameters that adapt using local and global information, respectively; the Hamiltonian Monte Carlo samplers, for example, define the step size as a fast parameter and the (co)variance as a slow parameter. The size of the the initial and final fast intervals and the initial size of the slow interval are all customizable, although user-specified values may be modified slightly in order to ensure alignment with the warmup period.

The motivation behind this partitioning of the warmup period is to allow for more robust adaptation. The stages are as follows.

1. In the initial fast interval the chain is allowed to converge towards the typical set,[1] with only parameters that can learn from local information adapted.
2. After this initial stage parameters that require global information, for example (co)variances, are estimated in a series of expanding, memoryless windows; often fast parameters will be adapted here as well.
3. Lastly, the fast parameters are allowed to adapt to the final update of the slow parameters.

These intervals may be controlled through the following configuration parameters, all of which must be positive integers:

Adaptation Parameters Table. *The parameters controlling adaptation and their default values.*

parameter	description	default
initial buffer	width of initial fast adaptation interval	75
term buffer	width of final fast adaptation interval	50
window	initial width of slow adaptation interval	25

Discretization-interval adaptation parameters

Stan's HMC algorithms utilize dual averaging Nesterov (2009) to optimize the step size.[2]

This warmup optimization procedure is extremely flexible and for completeness, Stan exposes each tuning option for dual averaging, using the notation of Hoffman and Gelman (2014). In practice, the efficacy of the optimization is sensitive to the value of these parameters, but we do not recommend changing the defaults without experience with the dual-averaging algorithm. For more information, see the discussion of dual averaging in Hoffman-Gelman:2014.

The full set of dual-averaging parameters are

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

[2] This optimization of step size during adaptation of the sampler should not be confused with running Stan's optimization method.

Step Size Adaptation Parameters Table *The parameters controlling step size adaptation, with constraints and default values.*

parameter	description	constraint	default
`delta`	target Metropolis acceptance rate	[0, 1]	0.8
`gamma`	adaptation regularization scale	(0, infty)	0.05
`kappa`	adaptation relaxation exponent	(0, infty)	0.75
`t_0`	adaptation iteration offset	(0, infty)	10

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

Step-size jitter

All implementations of HMC use numerical integrators requiring a step size (equivalently, discretization time interval). Stan allows the step size to be adapted or set explicitly. Stan also allows the step size to be "jittered" randomly during sampling to avoid any poor interactions with a fixed step size and regions of high curvature. The jitter is a proportion that may be added or subtracted, so the maximum amount of jitter is 1, which will cause step sizes to be selected in the range of 0 to twice the adapted step size. The default value is 0, producing no jitter.

Small step sizes can get HMC samplers unstuck that would otherwise get stuck with higher step sizes. The downside is that jittering below the adapted value will increase the number of leapfrog steps required and thus slow down iterations, whereas jittering above the adapted value can cause premature rejection due to simulation error in the Hamiltonian dynamics calculation. See Neal (2011) for further discussion of step-size jittering.

Euclidean metric

All HMC implementations in Stan utilize quadratic kinetic energy functions which are specified up to the choice of a symmetric, positive-definite matrix known as a *mass matrix* or, more formally, a *metric* Betancourt (2017).

If the metric is constant then the resulting implementation is known as *Euclidean* HMC. Stan allows a choice among three Euclidean HMC implementations,

- a unit metric (diagonal matrix of ones),

- a diagonal metric (diagonal matrix with positive diagonal entries), and
- a dense metric (a dense, symmetric positive definite matrix)

to be configured by the user.

If the metric is specified to be diagonal, then regularized variances are estimated based on the iterations in each slow-stage block (labeled II in the warmup adaptation stages figure). Each of these estimates is based only on the iterations in that block. This allows early estimates to be used to help guide warmup and then be forgotten later so that they do not influence the final covariance estimate.

If the metric is specified to be dense, then regularized covariance estimates will be carried out, regularizing the estimate to a diagonal matrix, which is itself regularized toward a unit matrix.

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

Warmup times and estimating the metric

The metric can compensate for linear (i.e. global) correlations in the posterior which can dramatically improve the performance of HMC in some problems. This requires knowing the global correlations.

In complex models, the global correlations are usually difficult, if not impossible, to derive analytically; for example, nonlinear model components convolve the scales of the data, so standardizing the data does not always help. Therefore, Stan estimates these correlations online with an adaptive warmup. In models with strong nonlinear (i.e. local) correlations this learning can be slow, even with regularization. This is ultimately why warmup in Stan often needs to be so long, and why a sufficiently long warmup can yield such substantial performance improvements.

Nonlinearity

The metric compensates for only linear (equivalently global or position-independent) correlations in the posterior. The hierarchical parameterizations, on the other hand, affect some of the nasty nonlinear (equivalently local or position-dependent) correlations common in hierarchical models.[3]

One of the biggest difficulties with dense metrics is the estimation of the metric itself which introduces a bit of a chicken-and-egg scenario; in order to estimate an appropriate metric for sampling, convergence is required, and in order to converge, an appropriate metric is required.

[3] In Riemannian HMC the metric compensates for nonlinear correlations.

Dense vs. diagonal metrics

Statistical models for which sampling is problematic are not typically dominated by linear correlations for which a dense metric can adjust. Rather, they are governed by more complex nonlinear correlations that are best tackled with better parameterizations or more advanced algorithms, such as Riemannian HMC.

Warmup times and curvature

MCMC convergence time is roughly equivalent to the autocorrelation time. Because HMC (and NUTS) chains tend to be lowly autocorrelated they also tend to converge quite rapidly.

This only applies when there is uniformity of curvature across the posterior, an assumption which is violated in many complex models. Quite often, the tails have large curvature while the bulk of the posterior mass is relatively well-behaved; in other words, warmup is slow not because the actual convergence time is slow but rather because the cost of an HMC iteration is more expensive out in the tails.

Poor behavior in the tails is the kind of pathology that can be uncovered by running only a few warmup iterations. By looking at the acceptance probabilities and step sizes of the first few iterations provides an idea of how bad the problem is and whether it must be addressed with modeling efforts such as tighter priors or reparameterizations.

NUTS and its configuration

The no-U-turn sampler (NUTS) automatically selects an appropriate number of leapfrog steps in each iteration in order to allow the proposals to traverse the posterior without doing unnecessary work. The motivation is to maximize the expected squared jump distance (see, e.g., Roberts, Gelman, and Gilks (1997)) at each step and avoid the random-walk behavior that arises in random-walk Metropolis or Gibbs samplers when there is correlation in the posterior. For a precise definition of the NUTS algorithm and a proof of detailed balance, see Hoffman and Gelman (2014).

NUTS generates a proposal by starting at an initial position determined by the parameters drawn in the last iteration. It then generates an independent standard normal random momentum vector. It then evolves the initial system both forwards and backwards in time to form a balanced binary tree. At each iteration of the NUTS algorithm the tree depth is increased by one, doubling the number of leapfrog steps and effectively doubles the computation time. The algorithm terminates in one of two ways, either

- the NUTS criterion (i.e., a U-turn in Euclidean space on a subtree) is satisfied

for a new subtree or the completed tree, or
- the depth of the completed tree hits the maximum depth allowed.

Rather than using a standard Metropolis step, the final parameter value is selected via multinomial sampling with a bias toward the second half of the steps in the trajectory Betancourt (2016b).[4]

Configuring the no-U-turn sample involves putting a cap on the depth of the trees that it evaluates during each iteration. This is controlled through a maximum depth parameter. The number of leapfrog steps taken is then bounded by 2 to the power of the maximum depth minus 1.

Both the tree depth and the actual number of leapfrog steps computed are reported along with the parameters in the output as `treedepth__` and `n_leapfrog__`, respectively. Because the final subtree may only be partially constructed, these two will always satisfy

$$2^{\text{treedepth}-1} - 1 \ < \ N_{\text{leapfrog}} \ \leq \ 2^{\text{treedepth}} - 1.$$

Tree depth is an important diagnostic tool for NUTS. For example, a tree depth of zero occurs when the first leapfrog step is immediately rejected and the initial state returned, indicating extreme curvature and poorly-chosen step size (at least relative to the current position). On the other hand, a tree depth equal to the maximum depth indicates that NUTS is taking many leapfrog steps and being terminated prematurely to avoid excessively long execution time. Taking very many steps may be a sign of poor adaptation, may be due to targeting a very high acceptance rate, or may simply indicate a difficult posterior from which to sample. In the latter case, reparameterization may help with efficiency. But in the rare cases where the model is correctly specified and a large number of steps is necessary, the maximum depth should be increased to ensure that that the NUTS tree can grow as large as necessary.

15.3. Sampling without parameters

In some situations, such as pure forward data simulation in a directed graphical model (e.g., where you can work down generatively from known hyperpriors to simulate parameters and data), there is no need to declare any parameters in Stan, the model block will be empty (and thus can be omitted), and all output quantities will be produced in the generated quantities block.

[4]Stan previously used slice sampling along the trajectory, following the original NUTS paper of Hoffman and Gelman (2014).

For example, to generate a sequence of N draws from a binomial with trials K and chance of success θ, the following program suffices.

```
data {
  real<lower=0, upper=1> theta;
  int<lower=0> K;
  int<lower=0> N;
}
generated quantities {
  array[N] int<lower=0, upper=K> y;
  for (n in 1:N) {
    y[n] = binomial_rng(K, theta);
  }
}
```

For this model, the sampler must be configured to use the fixed-parameters setting because there are no parameters. Without parameter sampling there is no need for adaptation and the number of warmup iterations should be set to zero.

Most models that are written to be sampled without parameters will not declare any parameters, instead putting anything parameter-like in the data block. Nevertheless, it is possible to include parameters for fixed-parameters sampling and initialize them in any of the usual ways (randomly, fixed to zero on the unconstrained scale, or with user-specified values). For example, `theta` in the example above could be declared as a parameter and initialized as a parameter.

15.4. General configuration options

Stan's interfaces provide a number of configuration options that are shared among the MCMC algorithms (this chapter), the optimization algorithms chapter, and the diagnostics chapter.

Random number generator

The random-number generator's behavior is fully determined by the unsigned seed (positive integer) it is started with. If a seed is not specified, or a seed of 0 or less is specified, the system time is used to generate a seed. The seed is recorded and included with Stan's output regardless of whether it was specified or generated randomly from the system time.

Stan also allows a chain identifier to be specified, which is useful when running multiple Markov chains for sampling. The chain identifier is used to advance the random number generator a very large number of random variates so that two

chains with different identifiers draw from non-overlapping subsequences of the random-number sequence determined by the seed. When running multiple chains from a single command, Stan's interfaces will manage the chain identifiers.

Replication

Together, the seed and chain identifier determine the behavior of the underlying random number generator. For complete reproducibility, every aspect of the environment needs to be locked down from the OS and version to the C++ compiler and version to the version of Stan and all dependent libraries.

Initialization

The initial parameter values for Stan's algorithms (MCMC, optimization, or diagnostic) may be either specified by the user or generated randomly. If user-specified values are provided, all parameters must be given initial values or Stan will abort with an error message.

User-defined initialization

If the user specifies initial values, they must satisfy the constraints declared in the model (i.e., they are on the constrained scale).

System constant zero initialization

It is also possible to provide an initialization of 0, which causes all variables to be initialized with zero values on the unconstrained scale. The transforms are arranged in such a way that zero initialization provides reasonable variable initializations for most parameters, such as 0 for unconstrained parameters, 1 for parameters constrained to be positive, 0.5 for variables to constrained to lie between 0 and 1, a symmetric (uniform) vector for simplexes, unit matrices for both correlation and covariance matrices, and so on.

System random initialization

Random initialization by default initializes the parameter values with values drawn at random from a $\mathsf{Uniform}(-2, 2)$ distribution. Alternatively, a value other than 2 may be specified for the absolute bounds. These values are on the unconstrained scale, so must be inverse transformed back to satisfy the constraints declared for parameters.

Because zero is chosen to be a reasonable default initial value for most parameters, the interval around zero provides a fairly diffuse starting point. For instance, unconstrained variables are initialized randomly in $(-2, 2)$, variables constrained to be positive are initialized roughly in $(0.14, 7.4)$, variables constrained to fall between 0 and 1 are initialized with values roughly in $(0.12, 0.88)$.

15.5. Divergent transitions

The Hamiltonian Monte Carlo algorithms (HMC and NUTS) simulate the trajectory of a fictitious particle representing parameter values when subject to a potential energy field, the value of which at a point is the negative log posterior density (up to a constant that does not depend on location). Random momentum is imparted independently in each direction, by drawing from a standard normal distribution. The Hamiltonian is defined to be the sum of the potential energy and kinetic energy of the system. The key feature of the Hamiltonian is that it is conserved along the trajectory the particle moves.

In Stan, we use the leapfrog algorithm to simulate the path of a particle along the trajectory defined by the initial random momentum and the potential energy field. This is done by alternating updates of the position based on the momentum and the momentum based on the position. The momentum updates involve the potential energy and are applied along the gradient. This is essentially a stepwise (discretized) first-order approximation of the trajectory. Leimkuhler and Reich (2004) provide details and error analysis for the leapfrog algorithm.

A divergence arises when the simulated Hamiltonian trajectory departs from the true trajectory as measured by departure of the Hamiltonian value from its initial value. When this divergence is too high,[5] the simulation has gone off the rails and cannot be trusted. The positions along the simulated trajectory after the Hamiltonian diverges will never be selected as the next draw of the MCMC algorithm, potentially reducing Hamiltonian Monte Carlo to a simple random walk and biasing estimates by not being able to thoroughly explore the posterior distribution. Betancourt (2016a) provides details of the theory, computation, and practical implications of divergent transitions in Hamiltonian Monte Carlo.

The Stan interfaces report divergences as warnings and provide ways to access which iterations encountered divergences. ShinyStan provides visualizations that highlight the starting point of divergent transitions to diagnose where the divergences arise in parameter space. A common location is in the neck of the funnel in a centered parameterization, an example of which is provided in the user's guide.

If the posterior is highly curved, very small step sizes are required for this gradient-based simulation of the Hamiltonian to be accurate. When the step size is too large (relative to the curvature), the simulation diverges from the true Hamiltonian. This definition is imprecise in the same way that stiffness for a differential equation is

[5] The current default threshold is a factor of 10^3, whereas when the leapfrog integrator is working properly, the divergences will be around 10^{-7} and do not compound due to the symplectic nature of the leapfrog integrator.

imprecise; both are defined by the way they cause traditional stepwise algorithms to diverge from where they should be.

The primary cause of divergent transitions in Euclidean HMC (other than bugs in the code) is highly varying posterior curvature, for which small step sizes are too inefficient in some regions and diverge in other regions. If the step size is too small, the sampler becomes inefficient and halts before making a U-turn (hits the maximum tree depth in NUTS); if the step size is too large, the Hamiltonian simulation diverges.

Diagnosing and eliminating divergences

In some cases, simply lowering the initial step size and increasing the target acceptance rate will keep the step size small enough that sampling can proceed. In other cases, a reparameterization is required so that the posterior curvature is more manageable; see the funnel example in the user's guide for an example.

Before reparameterization, it may be helpful to plot the posterior draws, highlighting the divergent transitions to see where they arise. This is marked as a divergent transition in the interfaces; for example, ShinyStan and RStan have special plotting facilities to highlight where divergent transitions arise.

16. Posterior Analysis

Stan uses Markov chain Monte Carlo (MCMC) techniques to generate samples from the posterior distribution for full Bayesian inference. Markov chain Monte Carlo (MCMC) methods were developed for situations in which it is not straightforward to make independent draws Metropolis et al. (1953).

Stan's variational inference algorithm provides draws from the variational approximation to the posterior which may be analyzed just as any other MCMC output, despite the fact that it is not actually a Markov chain.

Stan's Laplace algorithm produces a sample from a normal approximation centered at the mode of a distribution in the unconstrained space. If the mode is a maximum a posteriori (MAP) estimate, the samples provide an estimate of the mean and standard deviation of the posterior distribution. If the mode is a maximum likelihood estimate (MLE), the sample provides an estimate of the standard error of the likelihood.

16.1. Markov chains

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

$$p(\theta) = p(\theta^{(1)}) \prod_{n=2}^{N} p(\theta^{(n)} | \theta^{(n-1)}).$$

Stan uses Hamiltonian Monte Carlo to generate a next state in a manner described in the Hamiltonian Monte Carlo chapter.

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 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 target density $p(\theta)$ defined by a Stan program, which is typically a proper Bayesian posterior density $p(\theta|y)$ defined on the log scale up to a constant.

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 may be correlated or even anti-correlated, and thus the central limit theorem's bound on estimation error no longer applies. These problems are addressed in the next two sections.

Stan's posterior analysis tools compute a number of summary statistics, estimates, and diagnostics for Markov chain Monte Carlo (MCMC) samples. Stan's estimators and diagnostics are more robust in the face of non-convergence, antithetical sampling, and long-term Markov chain correlations than most of the other tools available. The algorithms Stan uses to achieve this are described in this chapter.

16.2. Convergence

By definition, a Markov chain generates samples from the target distribution only after it has converged to equilibrium (i.e., equilibrium is defined as being achieved when $p(\theta^{(n)})$ is the target density). The following point cannot be expressed strongly enough:

- In theory, *convergence is only guaranteed asymptotically* as the number of draws grows without bound.
- In practice, *diagnostics must be applied to monitor convergence* for the finite number of draws actually available.

16.3. Notation for samples, chains, and draws

To establish basic notation, suppose a target Bayesian posterior density $p(\theta|y)$ given real-valued vectors of parameters θ and real- and discrete-valued data y.[1]

An MCMC *sample* consists of a set of a sequence of M Markov chains, each consisting of an ordered sequence of N *draws* from the posterior.[2] The sample thus consists of $M \times N$ draws from the posterior.

[1] Using vectors simplifies high level exposition at the expense of collapsing structure.

[2] The structure is assumed to be rectangular; in the future, this needs to be generalized to ragged samples.

Potential scale reduction

One way to monitor whether a chain has converged to the equilibrium distribution is to compare its behavior to other randomly initialized chains. This is the motivation for the Gelman and Rubin (1992) potential scale reduction statistic, $\hat{R}$. The $\hat{R}$ statistic measures the ratio of the average variance of samples within each chain to the variance of the pooled samples across chains; if all chains are at equilibrium, these will be the same and $\hat{R}$ will be one. If the chains have not converged to a common distribution, the $\hat{R}$ statistic will be greater than one.

Gelman and Rubin's recommendation is that the independent Markov chains be initialized with diffuse starting values for the parameters and sampled until all values for $\hat{R}$ are below 1.1. Stan allows users to specify initial values for parameters and it is also able to draw diffuse random initializations automatically satisfying the declared parameter constraints.

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-chain variance* estimate is

$$B = \frac{N}{M-1} \sum_{m=1}^{M} (\bar{\theta}_m^{(\bullet)} - \bar{\theta}_\bullet^{(\bullet)})^2,$$

where

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

and

$$\bar{\theta}_\bullet^{(\bullet)} = \frac{1}{M} \sum_{m=1}^{M} \bar{\theta}_m^{(\bullet)}.$$

The *within-chain variance* is averaged over the chains,

$$W = \frac{1}{M} \sum_{m=1}^{M} s_m^2,$$

where

$$s_m^2 = \frac{1}{N-1} \sum_{n=1}^{N} (\theta_m^{(n)} - \bar{\theta}_m^{(\bullet)})^2.$$

The *variance estimator* is a mixture of the within-chain and cross-chain sample variances,

$$\widehat{\text{var}}^{+}(\theta|y) = \frac{N-1}{N} W + \frac{1}{N} B.$$

Finally, the *potential scale reduction statistic* is defined by

$$\hat{R} = \sqrt{\frac{\widehat{\text{var}}^{+}(\theta|y)}{W}}.$$

Split R-hat for detecting non-stationarity

Before Stan calculating the potential-scale-reduction statistic $\hat{R}$, each chain is split into two halves. This provides an additional means to detect non-stationarity in the individual 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.

Convergence is global

A question that often arises is whether it is acceptable to monitor convergence of only a subset of the parameters or generated quantities. The short answer is "no," but this is elaborated further in this section.

For example, consider the value `lp__`, which is the log posterior density (up to a constant).[3]

It is thus a mistake to declare convergence in any practical sense if `lp__` has not converged, because different chains are really in different parts of the space. Yet measuring convergence for `lp__` is particularly tricky, as noted below.

[3]The `lp__` value also represents the potential energy in the Hamiltonian system and is rate bounded by the randomly supplied kinetic energy each iteration, which follows a Chi-square distribution in the number of parameters.

Asymptotics and transience vs. equilibrium

Markov chain convergence is a global property in the sense that it does not depend on the choice of function of the parameters that is monitored. There is no hard cutoff between pre-convergence "transience" and post-convergence "equilibrium." What happens is that as the number of states in the chain approaches infinity, the distribution of possible states in the chain approaches the target distribution and in that limit the expected value of the Monte Carlo estimator of any integrable function converges to the true expectation. There is nothing like warmup here, because in the limit, the effects of initial state are completely washed out.

Multivariate convergence of functions

The $\hat{R}$ statistic considers the composition of a Markov chain and a function, and if the Markov chain has converged then each Markov chain and function composition will have converged. Multivariate functions converge when all of their margins have converged by the Cramer-Wold theorem.

The transformation from unconstrained space to constrained space is just another function, so does not effect convergence.

Different functions may have different autocorrelations, but if the Markov chain has equilibrated then all Markov chain plus function compositions should be consistent with convergence. Formally, any function that appears inconsistent is of concern and although it would be unreasonable to test every function, `lp__` and other measured quantities should at least be consistent.

The obvious difference in `lp__` is that it tends to vary quickly with position and is consequently susceptible to outliers.

Finite numbers of states

The question is what happens for finite numbers of states? If we can prove a strong geometric ergodicity property (which depends on the sampler and the target distribution), then one can show that there exists a finite time after which the chain forgets its initial state with a large probability. This is both the autocorrelation time and the warmup time. But even if you can show it exists and is finite (which is nigh impossible) you can't compute an actual value analytically.

So what we do in practice is hope that the finite number of draws is large enough for the expectations to be reasonably accurate. Removing warmup iterations improves the accuracy of the expectations but there is no guarantee that removing any finite number of samples will be enough.

Why inconsistent R-hat?

Firstly, as noted above, for any finite number of draws, there will always be some residual effect of the initial state, which typically manifests as some small (or large if the autocorrelation time is huge) probability of having a large outlier. Functions robust to such outliers (say, quantiles) will appear more stable and have better $\hat{R}$. Functions vulnerable to such outliers may show fragility.

Secondly, use of the $\hat{R}$ statistic makes very strong assumptions. In particular, it assumes that the functions being considered are Gaussian or it only uses the first two moments and assumes some kind of independence. The point is that strong assumptions are made that do not always hold. In particular, the distribution for the log posterior density (`lp__`) almost never looks Gaussian, instead it features long tails that can lead to large $\hat{R}$ even in the large N limit. Tweaks to $\hat{R}$, such as using quantiles in place of raw values, have the flavor of making the samples of interest more Gaussian and hence the $\hat{R}$ statistic more accurate.

Final words on convergence monitoring

"Convergence" is a global property and holds for all integrable functions at once, but employing the $\hat{R}$ statistic requires additional assumptions and thus may not work for all functions equally well.

Note that if you just compare the expectations between chains then we can rely on the Markov chain asymptotics for Gaussian distributions and can apply the standard tests.

16.4. Effective sample size

The second technical difficulty posed by MCMC methods is that the samples will typically be autocorrelated (or anticorrelated) within a chain. This increases the uncertainty of the estimation of posterior quantities of interest, such as means, variances, or quantiles; see Charles J. Geyer (2011).

Stan estimates an effective sample size for each parameter, which plays the role in the Markov chain Monte Carlo central limit theorem (MCMC CLT) as the number of independent draws plays in the standard central limit theorem (CLT).

Unlike most packages, the particular calculations used by Stan follow those for split-$\hat{R}$, which involve both cross-chain (mean) and within-chain calculations (autocorrelation); see Gelman et al. (2013).

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 the correlation between the two chains offset by t positions (i.e., a lag in time-series terminology). 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

$$\rho_t = \frac{1}{\sigma^2} \int_\Theta \theta^{(n)}\, \theta^{(n+t)}\, p(\theta)\, d\theta - \frac{\mu^2}{\sigma^2}.$$

The effective sample size of N samples generated by a process with autocorrelations ρ_t is defined by

$$N_{\text{eff}} \;=\; \frac{N}{\sum_{t=-\infty}^{\infty} \rho_t} \;=\; \frac{N}{1 + 2\sum_{t=1}^{\infty} \rho_t}.$$

For independent draws, the effective sample size is just the number of iterations. For correlated draws, the effective sample size will be lower than the number of iterations. For anticorrelated draws, the effective sample size can be larger than the number of iterations. In this latter case, MCMC can work better than independent sampling for some estimation problems. Hamiltonian Monte Carlo, including the no-U-turn sampler used by default in Stan, can produce anticorrelated draws if the posterior is close to Gaussian with little posterior correlation.

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 autocorrelations and split-$\hat{R}$ based effective sample size

estimator, based on multiple chains. As before, each chain θ_m will be assumed to be of length N.

Stan carries out the autocorrelation computations for all lags simultaneously using Eigen's fast Fourier transform (FFT) package with appropriate padding; see Charles J. Geyer (2011) for more detail on using FFT for autocorrelation calculations. The autocorrelation estimates $\hat{\rho}_{t,m}$ at lag t from multiple chains $m \in (1, \ldots, M)$ are combined with within-sample variance estimate W and multi-chain variance estimate $\widehat{\text{var}}^+$ introduced in the previous section to compute the combined autocorrelation at lag t as

$$\hat{\rho}_t = 1 - \frac{W - \frac{1}{M}\sum_{m=1}^{M} s_m^2 \hat{\rho}_{t,m}}{\widehat{\text{var}}^+}.$$

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, a typical truncated sum of $\hat{\rho}_t$ is used. Negative autocorrelations may occur only on odd lags and by summing over pairs starting from lag 0, the paired autocorrelation is guaranteed to be positive, monotone and convex modulo estimator noise Charles J. Geyer (1992), Charles J. Geyer (2011). Stan uses Geyer's initial monotone sequence criterion. The effective sample size estimator is defined as

$$\hat{N}_{\text{eff}} = \frac{M \cdot N}{\hat{\tau}},$$

where

$$\hat{\tau} = 1 + 2\sum_{t=1}^{2m+1} \hat{\rho}_t = -1 + 2\sum_{t'=0}^{m} \hat{P}_{t'},$$

where $\hat{P}_{t'} = \hat{\rho}_{2t'} + \hat{\rho}_{2t'+1}$. Initial positive sequence estimators is obtained by choosing the largest m such that $\hat{P}_{t'} > 0, \quad t' = 1, \ldots, m$. The initial monotone sequence is obtained by further reducing $\hat{P}_{t'}$ to the minimum of the preceding ones so that the estimated sequence is monotone.

Estimation of MCMC standard error

The posterior standard deviation of a parameter θ_n conditioned on observed data y is just the standard deviation of the posterior density $p(\theta_n|y)$. This is estimated by the standard deviation of the combined posterior draws across chains,

$$\hat{\sigma}_n = \text{sd}(\theta_n^{(1)}, \ldots, \theta_n^{(m)}).$$

The previous section showed how to estimate N_{eff} for a parameter θ_n based on multiple chains of posterior draws.

The mean of the posterior draws of θ_n

$$\hat{\theta}_n = \text{mean}(\theta_n^{(1)}, \ldots, \theta_n^{(m)})$$

is treated as an estimator of the true posterior mean,

$$\mathbb{E}[\theta_n \mid y] \;=\; \int_{-\infty}^{\infty} \theta \, p(\theta|y) \, \mathrm{d}\theta_n,$$

based the observed data y.

The standard error for the estimator $\hat{\theta}_n$ is given by the posterior standard deviation divided by the square root of the effective sample size. This standard error is itself estimated as $\hat{\sigma}_n / \sqrt{N_{\text{eff}}}$. The smaller the standard error, the closer the estimate $\hat{\theta}_n$ is expected to be to the true value. This is just the MCMC CLT applied to an estimator; see Charles J. Geyer (2011) for more details of the MCMC central limit theorem.

Thinning samples

In complex posteriors, draws are almost always positively correlated. In these situations, the autocorrelation at lag t, ρ_t, decreases as the lag, t, increases. In this situation, thinning the sample by keeping only every N-th draw will reduce the autocorrelation of the resulting chain. This is particularly useful if we need to save storage or re-use the draws for inference.

For instance, consider generating one thousand posterior draws in one of the following two ways.

- Generate 1000 draws after convergence and save all of them.
- Generate 10,000 draws after convergence and save every tenth draw.

Even though both produce a sample consisting one thousand draws, the second approach with thinning can produce a higher effective sample size when the draws are positively correlated. That's because the autocorrelation ρ_t for the thinned sequence is equivalent to ρ_{10t} in the unthinned sequence, so the sum of the autocorrelations will be lower and thus the effective sample size higher.

Now contrast the second approach above with the unthinned alternative,

- Generate 10,000 draws after convergence and save every draw.

This will typically have a higher effective sample than the thinned sample consisting of every tenth drawn. But the gap might not be very large. To summarize, *the only reason to thin a sample is to reduce memory requirements*.

If draws are anticorrelated, then thinning will increase correlation and reduce the overall effective sample size.

17. Optimization

Stan provides optimization algorithms which find modes of the density specified by a Stan program. Such modes may be used as parameter estimates or as the basis of approximations to a Bayesian posterior.

Stan provides three different optimizers, a Newton optimizer, and two related quasi-Newton algorithms, BFGS and L-BFGS; see Nocedal and Wright (2006) for thorough description and analysis of all of these algorithms. The L-BFGS algorithm is the default optimizer. Newton's method is the least efficient of the three, but has the advantage of setting its own stepsize.

17.1. General configuration

All of the optimizers have the option of including the the log absolute Jacobian determinant of inverse parameter transforms in the log probability computation. Without the Jacobian adjustment, optimization returns the maximum likelihood estimate (MLE), $\text{argmax}_\theta \; p(y|\theta)$, the value which maximizes the likelihood of the data given the parameters. Applying the Jacobian adjustment produces the maximum a posteriori estimate (MAP), that maximizes the value of the posterior density in the unconstrained space, $\text{argmax}_\theta \; p(y|\theta) \; p(\theta)$.

All of the optimizers are iterative and allow the maximum number of iterations to be specified; the default maximum number of iterations is 2000.

All of the optimizers are able to stream intermediate output reporting on their progress. Whether or not to save the intermediate iterations and stream progress is configurable.

17.2. BFGS and L-BFGS configuration

Convergence monitoring

Convergence monitoring in (L-)BFGS is controlled by a number of tolerance values, any one of which being satisfied causes the algorithm to terminate with a solution. Any of the convergence tests can be disabled by setting its corresponding tolerance parameter to zero. The tests for convergence are as follows.

Parameter convergence
The parameters θ_i in iteration i are considered to have converged with respect to tolerance `tol_param` if

$$||\theta_i - \theta_{i-1}|| < \texttt{tol\_param}.$$

Density convergence
The (unnormalized) log density $\log p(\theta_i|y)$ for the parameters θ_i in iteration i given data y is considered to have converged with respect to tolerance `tol_obj` if

$$|\log p(\theta_i|y) - \log p(\theta_{i-1}|y)| < \texttt{tol\_obj}.$$

The log density is considered to have converged to within relative tolerance `tol_rel_obj` if

$$\frac{|\log p(\theta_i|y) - \log p(\theta_{i-1}|y)|}{\max\left(|\log p(\theta_i|y)|, |\log p(\theta_{i-1}|y)|, 1.0\right)} < \texttt{tol\_rel\_obj} * \epsilon.$$

Gradient convergence
The gradient is considered to have converged to 0 relative to a specified tolerance `tol_grad` if

$$||g_i|| < \texttt{tol\_grad},$$

where ∇_θ is the gradient operator with respect to θ and $g_i = \nabla_\theta \log p(\theta|y)$ is the gradient at iteration i evaluated at $\theta^{(i)}$, the value on the i-th posterior iteration.

The gradient is considered to have converged to 0 relative to a specified relative tolerance `tol_rel_grad` if

$$\frac{g_i^T \hat{H}_i^{-1} g_i}{\max\left(|\log p(\theta_i|y)|, 1.0\right)} < \texttt{tol\_rel\_grad} * \epsilon,$$

where $\hat{H}_i$ is the estimate of the Hessian at iteration i, $|u|$ is the absolute value (L1 norm) of u, $||u||$ is the vector length (L2 norm) of u, and $\epsilon \approx 2e-16$ is machine precision.

Initial step size

The initial step size parameter α for BFGS-style optimizers may be specified. If the first iteration takes a long time (and requires a lot of function evaluations) initialize α to be the roughly equal to the α used in that first iteration. The default value is intentionally small, 0.001, which is reasonable for many problems but might be too large or too small depending on the objective function and initialization. Being too big or too small just means that the first iteration will take longer (i.e., require more gradient evaluations) before the line search finds a good step length. It's not a critical parameter, but for optimizing the same model multiple times (as you tweak things or with different data), being able to tune α can save some real time.

L-BFGS history size

L-BFGS has a command-line argument which controls the size of the history it uses to approximate the Hessian. The value should be less than the dimensionality of the parameter space and, in general, relatively small values (5–10) are sufficient; the default value is 5.

If L-BFGS performs poorly but BFGS performs well, consider increasing the history size. Increasing history size will increase the memory usage, although this is unlikely to be an issue for typical Stan models.

17.3. Writing models for optimization

Constrained vs. unconstrained parameters

For constrained optimization problems, for instance, with a standard deviation parameter σ constrained so that $\sigma > 0$, it can be much more efficient to declare a parameter `sigma` with no constraints. This allows the optimizer to easily get close to 0 without having to tend toward $-\infty$ on the $\log \sigma$ scale.

With unconstrained parameterizations of parameters with constrained support, it is important to provide a custom initialization that is within the support. For example, declaring a vector

```
vector[M] sigma;
```

and using the default random initialization which is $\text{Uniform}(-2, 2)$ on the unconstrained scale means that there is only a 2^{-M} chance that the initialization will be within support.

For any given optimization problem, it is probably worthwhile trying the program both ways, with and without the constraint, to see which one is more efficient.

18. Pathfinder

Stan supports the Pathfinder algorithm Zhang et al. (2022). Pathfinder is a variational method for approximately sampling from differentiable log densities. Starting from a random initialization, Pathfinder locates normal approximations to the target density along a quasi-Newton optimization path, with local covariance estimated using the negative inverse Hessian estimates produced by the LBFGS optimizer. Pathfinder returns draws from the Gaussian approximation with the lowest estimated Kullback-Leibler (KL) divergence to the true posterior.

Stan provides two versions of the Pathfinder algorithm: single-path Pathfinder and multi-path Pathfinder. Single-path Pathfinder generates a set of approximate draws from one run of the basic Pathfinder algorithm. Multi-path Pathfinder uses importance resampling over the draws from multiple runs of Pathfinder. This better matches non-normal target densities and also mitigates the problem of L-BFGS getting stuck at local optima or in saddle points on plateaus. Compared to ADVI and short dynamic HMC runs, Pathfinder requires one to two orders of magnitude fewer log density and gradient evaluations, with greater reductions for more challenging posteriors. While the evaluations in Zhang et al. (2022) found that single-path and multi-path Pathfinder outperform ADVI for most of the models in the PosteriorDB evaluation set, we recognize the need for further experiments on a wider range of models.

19. Variational Inference

Stan implements an automatic variational inference algorithm, called Automatic Differentiation Variational Inference (ADVI) Kucukelbir et al. (2017). In this chapter, we describe the specifics of how ADVI maximizes the variational objective.

19.1. Stochastic gradient ascent

ADVI optimizes the ELBO in the real-coordinate space using stochastic gradient ascent. We obtain noisy (yet unbiased) gradients of the variational objective using automatic differentiation and Monte Carlo integration. The algorithm ascends these gradients using an adaptive stepsize sequence. We evaluate the ELBO also using Monte Carlo integration and measure convergence similar to the relative tolerance scheme in Stan's optimization feature.

Monte Carlo approximation of the ELBO

ADVI uses Monte Carlo integration to approximate the variational objective function, the ELBO. The number of draws used to approximate the ELBO is denoted by `elbo_samples`. We recommend a default value of 100, as we only evaluate the ELBO every `eval_elbo` iterations, which also defaults to 100.

Monte Carlo approximation of the gradients

ADVI uses Monte Carlo integration to approximate the gradients of the ELBO. The number of draws used to approximate the gradients is denoted by `grad_samples`. We recommend a default value of 1, as this is the most efficient. It also a very noisy estimate of the gradient, but stochastic gradient ascent is capable of following such gradients.

Adaptive stepsize sequence

ADVI uses a finite-memory version of adaGrad Duchi, Hazan, and Singer (2011). This has a single parameter that we expose, denoted `eta`. We now have a warmup adaptation phase that selects a good value for `eta`. The procedure does a heuristic search over `eta` values that span 5 orders of magnitude.

Assessing convergence

ADVI tracks the progression of the ELBO through the stochastic optimization. Specifically, ADVI heuristically determines a rolling window over which it computes the average and the median change of the ELBO. Should either number fall

below a threshold, denoted by `tol_rel_obj`, we consider the algorithm to have converged. The change in ELBO is calculated the same way as in Stan's optimization module.

20. Laplace Approximation

Stan provides a Laplace approximation algorithm which can be used to obtain samples from an approximated posterior. The Laplace approximation works in the unconstrained space, so that if there are constrained parameters, the normal approximation is centered at the mode in the unconstrained space and then the implemented method transforms the normal approximation sample to the constrained space before outputting them.

Given the estimate of the mode $\widehat{\theta}$, the Hessian $H(\widehat{\theta})$ is computed using central finite differences of the model functor. Next the algorithm computes the Cholesky factor of the negative inverse Hessian:

$R^{-1} = \text{chol}(-H(\widehat{\theta}))\backslash\mathbf{1}$.

Each draw is generated on the unconstrained scale by sampling

$\theta^{\text{std}(m)} \sim \text{normal}(0, \text{I})$

and defining draw m to be

$\theta^{(m)} = \widehat{\theta} + R^{-1} \cdot \theta^{\text{std}(m)}$

Finally, each $\theta^{(m)}$ is transformed back to the constrained scale.

The one-time computation of the Cholesky factor incurs a high constant overhead of $\mathcal{O}(N^3)$ in N dimensions. It also requires $2N$ gradient calculations to use as the basis, which scales at best as $\mathcal{O}(N^2)$ and is worse for models whose gradient calculation is super-linear in dimension. The algorithm also has a high per-draw overhead, requiring N standard normal pseudorandom numbers and $\mathcal{O}(N^2)$ per draw (to multiply by the Cholesky factor). For M draws, the total cost is proportional to $\mathcal{O}(N^3 + M \cdot N^2)$.

21. Diagnostic Mode

Stan's diagnostic mode runs a Stan program with data, initializing parameters either randomly or with user-specified initial values, and then evaluates the log probability and its gradients. The gradients computed by the Stan program are compared to values calculated by finite differences.

Diagnostic mode may be configured with two parameters.

Diagnostic Mode Configuration Table. *The diagnostic model configuration parameters, constraints, and default values.*

parameter	description	constraints	default
`epsilon`	finite difference size	(0, infty)	1e–6
`error`	error threshold for matching	(0, infty)	1e–6

If the difference between the Stan program's gradient value and that calculated by finite difference is higher than the specified threshold, the argument will be flagged.

21.1. Diagnostic mode output

Diagnostic mode prints the log posterior density (up to a proportion) calculated by the Stan program for the specified initial values. For each parameter, it prints the gradient at the initial parameter values calculated by Stan's program and by finite differences over Stan's program for the log probability.

Unconstrained scale

The output is for the variable values and their gradients are on the unconstrained scale, which means each variable is a vector of size corresponding to the number of unconstrained variables required to define it. For example, an $N \times N$ correlation matrix, requires $\binom{N}{2}$ unconstrained parameters. The transformations from constrained to unconstrained parameters are based on the constraints in the parameter declarations and described in the reference manual chapter on transforms.

Includes Jacobian

The log density includes the Jacobian adjustment implied by the constraints declared on variables. The Jacobian adjustment for constrained parameter transforms may be

turned off for optimization, but there is as of yet no way to turn it off in diagnostic mode.

21.2. Configuration options

The general configuration options for diagnostics are the same as those for MCMC. Initial values may be specified, or they may be drawn at random. Setting the random number generator will only have an effect if a random initialization is specified.

21.3. Speed warning and data trimming

Due to the application of finite differences, the computation time grows linearly with the number of parameters. This can be require a very long time, especially in models with latent parameters that grow with the data size. It can be helpful to diagnose a model with smaller data sizes in such cases.

Part III

Usage

22. Reproducibility

Floating point operations on modern computers are notoriously difficult to replicate because the fundamental arithmetic operations, right down to the IEEE 754 encoding level, are not fully specified. The primary problem is that the precision of operations varies across different hardware platforms and software implementations.

Stan is designed to allow full reproducibility. However, this is only possible up to the external constraints imposed by floating point arithmetic.

Stan results will only be exactly reproducible if *all* of the following components are *identical*:

- Stan version
- Stan interface (RStan, PyStan, CmdStan) and version, plus version of interface language (R, Python, shell)
- versions of included libraries (Boost and Eigen)
- operating system version
- computer hardware including CPU, motherboard and memory
- C++ compiler, including version, compiler flags, and linked libraries
- same configuration of call to Stan, including random seed, chain ID, initialization and data

It doesn't matter if you use a stable release version of Stan or the version with a particular Git hash tag. The same goes for all of the interfaces, compilers, and so on. The point is that if any of these moving parts changes in some way, floating point results may change.

Concretely, if you compile a single Stan program using the same CmdStan code base, but changed the optimization flag (`-O3` vs. `-O2` or `-O0`), the two programs may not return the identical stream of results. Thus it is very hard to guarantee reproducibility on externally managed hardware, like in a cluster or even a desktop managed by an IT department or with automatic updates turned on.

If, however, you compiled a Stan program today using one set of flags, took the computer away from the internet and didn't allow it to update anything, then came back in a decade and recompiled the Stan program in the same way, you should get the same results.

The data needs to be the same down to the bit level. For example, if you are running in RStan, Rcpp handles the conversion between R's floating point numbers and C++ doubles. If Rcpp changes the conversion process or use different types, the results are not guaranteed to be the same down to the bit level.

The compiler and compiler settings can also be an issue. There is a nice discussion of the issues and how to control reproducibility in Intel's proprietary compiler by Corden and Kreitzer (2014).

22.1. Notable changes across versions

As noted above, there is no guarantee that the same results will be reproducible between two different versions of Stan, even if the same settings and environment are used.

However, there are occassionally notable changes which would affect many if not all users, and these are noted here. The absence of a version from this list still *does not* guarantee exact reproducibility between it and other versions.

- Stan 2.28 changed the default chain ID for MCMC from `0` to `1`. Users who had set a seed but not a chain ID would observe completely different outputs.
- Stan 2.35 changed the default pseudo-random number generator used by the Stan algorithms. There is no relationship between seeds in versions pre-2.35 and version 2.35 and on.

23. Licenses and Dependencies

Stan and its dependent libraries, are distributed under generous, freedom-respecting licenses approved by the Open Source Initiative.

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 specifies the licenses for the libraries on which Stan's math library, language, and algorithms depend. The last tool mentioned, Google Test, is only used for testing and is not needed to run Stan.

23.1. Stan license

Stan is distributed under

- BSD 3-clause license (BSD New)

The copyright holder of each contribution is the developer or his or her assignee.[1]

23.2. Boost license

Stan uses the Boost library for template metaprograms, traits programs, the parser, and various numerical libraries for special functions, probability functions, and random number generators. Boost is distributed under the

- Boost Software License version 1.0

The copyright for each Boost package is held by its developers or their assignees.

23.3. Eigen license

Stan uses the Eigen library for matrix arithmetic and linear algebra. Eigen is distributed under the

- Mozilla Public License, version 2

The copyright of Eigen is owned jointly by its developers or their assignees.

[1] Universities or companies often own the copyright of computer programs developed by their employees.

23.4. SUNDIALS license

Stan uses the SUNDIALS package for solving differential equations. SUNDIALS is distributed under the

- BSD 3-clause license (BSD New)

The copyright of SUNDIALS is owned by Lawrence Livermore National Security Lab.

23.5. Threaded Building Blocks (TBB) License

Stan uses the Threaded Building Blocks (TBB) library for parallel computations. TBB is distributed under the

- Apache License, version 2

The copyright of TBB is owned by Intel Corporation.

23.6. 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 3-clause license (BSD New)

The copyright of Google Test is owned by Google, Inc.

References

Betancourt, Michael. 2010. "Cruising the Simplex: Hamiltonian Monte Carlo and the Dirichlet Distribution." *arXiv* 1010.3436. http://arxiv.org/abs/1010.3436.

———. 2016a. "Diagnosing Suboptimal Cotangent Disintegrations in Hamiltonian Monte Carlo." *arXiv* 1604.00695. https://arxiv.org/abs/1604.00695.

———. 2016b. "Identifying the Optimal Integration Time in Hamiltonian Monte Carlo." *arXiv* 1601.00225. https://arxiv.org/abs/1601.00225.

———. 2017. "A Conceptual Introduction to Hamiltonian Monte Carlo." *arXiv* 1701.02434. https://arxiv.org/abs/1701.02434.

Betancourt, Michael, and Mark Girolami. 2013. "Hamiltonian Monte Carlo for Hierarchical Models." *arXiv* 1312.0906. http://arxiv.org/abs/1312.0906.

Corden, Martyn J., and David Kreitzer. 2014. "Consistency of Floating-Point Results Using the Intel Compiler or Why Doesn't My Application Always Give the Same Answer?" Intel Corporation. https://software.intel.com/en-us/articles/consistency-of-floating-point-results-using-the-intel-compiler.

Duchi, John, Elad Hazan, and Yoram Singer. 2011. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization." *The Journal of Machine Learning Research* 12: 2121–59.

Gelman, Andrew, J. B. Carlin, Hal S. Stern, David B. Dunson, Aki Vehtari, and Donald B. Rubin. 2013. *Bayesian Data Analysis*. Third Edition. London: Chapman & Hall / CRC Press.

Gelman, Andrew, and Jennifer Hill. 2007. *Data Analysis Using Regression and Multilevel-Hierarchical Models*. Cambridge, United Kingdom: Cambridge University Press.

Gelman, Andrew, and Donald B. Rubin. 1992. "Inference from Iterative Simulation Using Multiple Sequences." *Statistical Science* 7 (4): 457–72.

Geyer, Charles J. 1992. "Practical Markov Chain Monte Carlo." *Statistical Science*, 473–83.

Geyer, Charles J. 2011. "Introduction to Markov Chain Monte Carlo." In *Handbook of Markov Chain Monte Carlo*, edited by Steve Brooks, Andrew Gelman, Galin L. Jones, and Xiao-Li Meng, 3–48. Chapman; Hall/CRC.

Hoffman, Matthew D., and Andrew Gelman. 2014. "The No-U-Turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo." *Journal of Machine Learning Research* 15: 1593–623. http://jmlr.org/papers/v15/hoffman14a.html.

Kucukelbir, Alp, Dustin Tran, Rajesh Ranganath, Andrew Gelman, and David M Blei. 2017. "Automatic Differentiation Variational Inference." *Journal of Machine Learning Research*.

Leimkuhler, Benedict, and Sebastian Reich. 2004. *Simulating Hamiltonian Dynamics*. Cambridge: Cambridge University Press.

Lewandowski, Daniel, Dorota Kurowicka, and Harry Joe. 2009. "Generating Random Correlation Matrices Based on Vines and Extended Onion Method." *Journal of Multivariate Analysis* 100: 1989–2001.

Metropolis, N., A. Rosenbluth, M. Rosenbluth, M. Teller, and E. Teller. 1953. "Equations of State Calculations by Fast Computing Machines." *Journal of Chemical Physics* 21: 1087–92.

Muller, Mervin E. 1959. "A Note on a Method for Generating Points Uniformly on n-Dimensional Spheres." *Commun. ACM* 2 (4): 19–20. https://doi.org/10.1145/377939.377946.

Neal, Radford. 2011. "MCMC Using Hamiltonian Dynamics." In *Handbook of Markov Chain Monte Carlo*, edited by Steve Brooks, Andrew Gelman, Galin L. Jones, and Xiao-Li Meng, 116–62. Chapman; Hall/CRC.

Nesterov, Y. 2009. "Primal-Dual Subgradient Methods for Convex Problems." *Mathematical Programming* 120 (1): 221–59.

Nocedal, Jorge, and Stephen J. Wright. 2006. *Numerical Optimization*. Second. Berlin: Springer-Verlag.

Roberts, G. O., Andrew Gelman, and Walter R. Gilks. 1997. "Weak Convergence and Optimal Scaling of Random Walk Metropolis Algorithms." *Annals of Applied Probability* 7 (1): 110–20.

Zhang, Lu, Bob Carpenter, Andrew Gelman, and Aki Vehtari. 2022. "Pathfinder: Parallel Quasi-Newton Variational Inference." *Journal of Machine Learning Research* 23 (306): 1–49. http://jmlr.org/papers/v23/21-0889.html.