Stan User's Guide

Version 2.22

Stan Development Team

Contents

Overview

About this user's guide

This is the official user's guide for Stan. It provides example models and programming techniques for coding statistical models in Stan.

- Part 1 gives Stan code and discussions for several important classes of models.
- Part 2 discusses various general Stan programming techniques that are not tied to any particular model.
- The appendices provide a style guide and advice for users of BUGS and JAGS.

In addition to this user's guide, there are two reference manuals for the Stan language and algorithms. The *Stan Reference Manual* specifies the Stan programming language and inference algorithms. The *Stan Functions Reference* specifies the functions built into the Stan programming language.

There is also a separate installation and getting started guide for each of the Stan interfaces (R, Python, Julia, Stata, MATLAB, Mathematica, and command line).

We recommend working through this guide using the textbooks *Bayesian Data Analysis* and *Statistical Rethinking: A Bayesian Course with Examples in R and Stan* as references on the concepts, and using the *Stan Reference Manual* when necessary to clarify programming issues.

Web resources

Stan is an open-source software project, resources for which are hosted on various web sites:

- The Stan Web Site organizes all of the resources for the Stan project for users and developers. It contains links to the official Stan releases, source code, installation instructions, and full documentation, including the latest version of this manual, the user's guide and the getting started guide for each interface, tutorials, case studies, and reference materials for developers.
- The Stan Forums provide structured message boards for questions, discussion, and announcements related to Stan for both users and developers.

- The Stan GitHub Organization hosts all of Stan's code, documentation, wikis, and web site, as well as the issue trackers for bug reports and feature requests and interactive code review for pull requests.

Acknowledgements

The Stan project could not exist without developers, users, and funding. Stan is a highly collaborative project. The individual contributions of the Stan developers to code is tracked through GitHub and to the design conversation in the Wikis and forums.

Users have made extensive contributions to documentation in the way of case studies, tutorials and even books. They have also reported numerous bugs in both the code and documentation.

Stan has been funded through grants for Stan and its developers, through in-kind donations in the form of companies contributing developer time to Stan and individuals contributing their own time to Stan, and through donations to the open-source scientific software non-profit NumFOCUS. For details of direct funding for the project, see the web site and project pages of the Stan developers.

Part 1. Example Models

In this part of the book, we survey a range of example models, with the goal of illustrating how to code them efficiently in Stan.

1. Regression Models

Stan supports regression models from simple linear regressions to multilevel generalized linear models.

1.1. Linear Regression

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

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

This is equivalent to the following sampling involving the residual,

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

and reducing still further, to

$$y_n \sim \text{normal}(\alpha + \beta X_n, \, \sigma).$$

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

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

There are `N` observations, each with predictor `x[n]` and outcome `y[n]`. The intercept and slope parameters are `alpha` and `beta`. The model assumes a normally distributed

noise term with scale `sigma`. This model has improper priors for the two regression coefficients.

Matrix Notation and Vectorization

The sampling statement in the previous model is vectorized, with

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

providing the same model as the unvectorized version,

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

In addition to being more concise, the vectorized form is much faster.[1]

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

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

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

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

```
  y ~ normal(x * beta + alpha, sigma);  // likelihood
}
```

The constraint `lower=0` in the declaration of `sigma` constrains the value to be greater than or equal to 0. With no prior in the model block, the effect is an improper prior on non-negative real numbers. Although a more informative prior may be added, improper priors are acceptable as long as they lead to proper posteriors.

In the model above, `x` is an $N \times K$ matrix of predictors and `beta` a K-vector of coefficients, so `x * beta` is an N-vector of predictions, one for each of the N data items. These predictions line up with the outcomes in the N-vector `y`, so the entire model may be written using matrix arithmetic as shown. It would be possible to include a column of ones the data matrix `x` and remove the `alpha` parameter.

The sampling statement in the model above is just a more efficient, vector-based approach to coding the model with a loop, as in the following statistically equivalent model.

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

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

Intercepts as Inputs

In the model formulation

```
y ~ normal(x * beta, sigma);
```

there is no longer an intercept coefficient `alpha`. Instead, we have assumed that the first column of the input matrix `x` is a column of 1 values. This way, `beta[1]` plays the role of the intercept. If the intercept gets a different prior than the slope terms, then it would be clearer to break it out. It is also slightly more efficient in its explicit form with the intercept variable singled out because there's one fewer multiplications; it should not make that much of a difference to speed, though, so the choice should be based on clarity.

1.2. The QR Reparameterization

In the previous example, the linear predictor can be written as $\eta = x\beta$, where η is a N-vector of predictions, x is a $N \times K$ matrix, and β is a K-vector of coefficients. Presuming $N \geq K$, we can exploit the fact that any design matrix x can be decomposed using the thin QR decomposition into an orthogonal matrix Q and an upper-triangular matrix R, i.e. $x = QR$.

The functions `qr_thin_Q` and `qr_thin_R` implement the thin QR decomposition, which is to be preferred to the fat QR decomposition that would be obtained by using `qr_Q` and `qr_R`, as the latter would more easily run out of memory (see the Stan Functions Reference for more information on the `qr_thin_Q` and `qr_thin_R` functions). In practice, it is best to write $x = Q^* R^*$ where $Q^* = Q * \sqrt{n-1}$ and $R^* = \frac{1}{\sqrt{n-1}} R$. Thus, we can equivalently write $\eta = x\beta = QR\beta = Q^* R^* \beta$. If we let $\theta = R^* \beta$, then we have $\eta = Q^* \theta$ and $\beta = {R^*}^{-1} \theta$. In that case, the previous Stan program becomes

```
data {
  int<lower=0> N;   // number of data items
  int<lower=0> K;   // number of predictors
  matrix[N, K] x;   // predictor matrix
  vector[N] y;      // outcome vector
}
transformed data {
  matrix[N, K] Q_ast;
  matrix[K, K] R_ast;
  matrix[K, K] R_ast_inverse;
  // thin and scale the QR decomposition
  Q_ast = qr_thin_Q(x) * sqrt(N - 1);
  R_ast = qr_thin_R(x) / sqrt(N - 1);
  R_ast_inverse = inverse(R_ast);
}
parameters {
  real alpha;           // intercept
  vector[K] theta;      // coefficients on Q_ast
  real<lower=0> sigma;  // error scale
}
model {
  y ~ normal(Q_ast * theta + alpha, sigma);  // likelihood
}
```

```
generated quantities {
  vector[K] beta;
  beta = R_ast_inverse * theta; // coefficients on x
}
```

Since this Stan program generates equivalent predictions for y and the same posterior distribution for α, β, and σ as the previous Stan program, many wonder why the version with this QR reparameterization performs so much better in practice, often both in terms of wall time and in terms of effective sample size. The reasoning is threefold:

1. The columns of Q^* are orthogonal whereas the columns of x generally are not. Thus, it is easier for a Markov Chain to move around in θ-space than in β-space.
2. The columns of Q^* have the same scale whereas the columns of x generally do not. Thus, a Hamiltonian Monte Carlo algorithm can move around the parameter space with a smaller number of larger steps
3. Since the covariance matrix for the columns of Q^* is an identity matrix, θ typically has a reasonable scale if the units of y are also reasonable. This also helps HMC move efficiently without compromising numerical accuracy.

Consequently, this QR reparameterization is recommended for linear and generalized linear models in Stan whenever $K > 1$ and you do not have an informative prior on the location of β. It can also be worthwhile to subtract the mean from each column of x before obtaining the QR decomposition, which does not affect the posterior distribution of θ or β but does affect α and allows you to interpret α as the expectation of y in a linear model.

1.3. Priors for Coefficients and Scales

See our general discussion of priors for tips on priors for parameters in regression models.

Later sections discuss univariate hierarchical priors and multivariate hierarchical priors, as well as priors used to identify models.

However, as described in QR-reparameterization section, if you do not have an informative prior on the *location* of the regression coefficients, then you are better off reparameterizing your model so that the regression coefficients are a generated quantity. In that case, it usually does not matter much what prior is used on on the

reparameterized regression coefficients and almost any weakly informative prior that scales with the outcome will do.

1.4. Robust Noise Models

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

```
data {
  ...
  real<lower=0> nu;
}
...
model {
  y ~ student_t(nu, alpha + beta * x, sigma);
}
```

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

1.5. Logistic and Probit Regression

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

A logistic regression model with one predictor and an intercept is coded as follows.

```
data {
  int<lower=0> N;
  vector[N] x;
  int<lower=0,upper=1> y[N];
}
parameters {
  real alpha;
  real beta;
}
model {
```

```
  y ~ bernoulli_logit(alpha + beta * x);
}
```

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

Logistic regression is a kind of generalized linear model with binary outcomes and the log odds (logit) link function, defined by

$$\text{logit}(v) = \log\left(\frac{v}{1-v}\right).$$

The inverse of the link function appears in the model:

$$\text{logit}^{-1}(u) = \texttt{inv\_logit}(u) = \frac{1}{1+\exp(-u)}.$$

The model formulation above uses the logit-parameterized version of the Bernoulli distribution, which is defined by

$$\texttt{bernoulli\_logit}\left(y \mid \alpha\right) = \texttt{bernoulli}\left(y \mid \text{logit}^{-1}(\alpha)\right).$$

The formulation is also vectorized in the sense that `alpha` and `beta` are scalars and `x` is a vector, so that `alpha + beta * x` is a vector. The vectorized formulation is equivalent to the less efficient version

```
for (n in 1:N)
  y[n] ~ bernoulli_logit(alpha + beta * x[n]);
```

Expanding out the Bernoulli logit, the model is equivalent to the more explicit, but less efficient and less arithmetically stable

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

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

$$\Phi(x) = \int_{-\infty}^{x} \mathsf{normal}\left(y \mid 0, 1\right) \, \mathrm{d}y.$$

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

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

A fast approximation to the cumulative standard normal distribution function Φ is implemented in Stan as the function `Phi_approx`.[2] The approximate probit regression model may be coded with the following.

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

1.6. Multi-Logit Regression

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

```
data {
  int K;
  int N;
  int D;
  int y[N];
  matrix[N, D] x;
}
parameters {
  matrix[D, K] beta;
}
model {
  matrix[N, K] x_beta = x * beta;

  to_vector(beta) ~ normal(0, 5);

  for (n in 1:N)
    y[n] ~ categorical_logit(x_beta[n]');
}
```

The prior on `beta` is coded in vectorized form. As of Stan 2.18, the categorical-logit distribution is not vectorized for parameter arguments, so the loop is required. The matrix multiplication is pulled out to define a local variable for all of the predictors for efficiency. Like the Bernoulli-logit, the categorical-logit distribution applies

[2] The `Phi_approx` function is a rescaled version of the inverse logit function, so while the scale is roughly the same Φ, the tails do not match.

softmax internally to convert an arbitrary vector to a simplex,

$$\texttt{categorical\_logit}\,(y \mid \alpha) = \texttt{categorical}\,(y \mid \texttt{softmax}(\alpha))\,,$$

where

$$\texttt{softmax}(u) = \exp(u)/\,\mathrm{sum}\,(\exp(u))\,.$$

The categorical distribution with log-odds (logit) scaled parameters used above is equivalent to writing

```
y[n] ~ categorical(softmax(x[n] * beta));
```

Constraints on Data Declarations

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

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

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

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

Identifiability

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

An alternative is to use $(K-1)$-vectors by fixing one of them to be zero. The partially known parameters section discusses how to mix constants and parameters in a vector. In the multi-logit case, the parameter block would be redefined to use $(K-1)$-vectors

```
parameters {
  matrix[K - 1, D] beta_raw;
}
```

and then these are transformed to parameters to use in the model. First, a transformed data block is added before the parameters block to define a row vector of zero values,

```
transformed data {
  row_vector[D] zeros = rep_row_vector(0, D);
}
```

which can then be appended to `beta_row` to produce the coefficient matrix `beta`,

```
transformed parameters {
  matrix[K, D] beta;
  beta = append_row(beta_raw, zeros);
}
```

The `rep_row_vector(0, D)` call creates a row vector of size `D` with all entries set to zero. The derived matrix `beta` is then defined to be the result of appending the row-vector `zeros` as a new row at the end of `beta_raw`; the row vector `zeros` is defined as transformed data so that it doesn't need to be constructed from scratch each time it is used.

This is not the same model as using K-vectors as parameters, because now the prior only applies to $(K-1)$-vectors. In practice, this will cause the maximum likelihood solutions to be different and also the posteriors to be slightly different when taking priors centered around zero, as is typical for regression coefficients.

1.7. Parameterizing Centered Vectors

It is often convenient to define a parameter vector β that is centered in the sense of satisfying the sum-to-zero constraint,

$$\sum_{k=1}^{K} \beta_k = 0.$$

Such a parameter vector may be used to identify a multi-logit regression parameter vector (see the multi-logit section for details), or may be used for ability or difficulty parameters (but not both) in an IRT model (see the item-response model section for details).

$K - 1$ Degrees of Freedom

There is more than one way to enforce a sum-to-zero constraint on a parameter vector, the most efficient of which is to define the K-th element as the negation of the sum of the elements 1 through $K - 1$.

```
parameters {
  vector[K-1] beta_raw;
  ...
transformed parameters {
  vector[K] beta = append_row(beta_raw, -sum(beta_raw));
  ...
```

Placing a prior on `beta_raw` in this parameterization leads to a subtly different posterior than that resulting from the same prior on `beta` in the original parameterization without the sum-to-zero constraint. Most notably, a simple prior on each component of `beta_raw` produces different results than putting the same prior on each component of an unconstrained K-vector `beta`. For example, providing a $\mathsf{normal}(0, 5)$ prior on `beta` will produce a different posterior mode than placing the same prior on `beta_raw`.

Marginal distribution of sum-to-zero components

On the Stan forums, Aaron Goodman provided the following code to produce a prior with standard normal marginals on the components of `beta`,

```
model {
  beta ~ normal(0, inv(sqrt(1 - inv(K))));
```

```
  ...
```

The components are not independent, as they must sum zero. No Jacobian is required because summation and negation are linear operations (and thus have constant Jacobians).

To generate distributions with marginals other than standard normal, the resulting `beta` may be scaled by some factor `sigma` and translated to some new location `mu`.

QR Decomposition

Aaron Goodman, on the Stan forums, also provided this approach, which calculates a QR decomposition in the transformed data block, then uses it to transform to a sum-to-zero parameter `x`,

```
transformed data{
  matrix[K, K] A = diag_matrix(rep_vector(1,K));
  matrix[K, K-1] A_qr;
  for (i in 1:K-1) A[K,i] = -1;
  A[K,K] = 0;
  A_qr = qr_Q(A)[ , 1:(K-1)];
}
parameters {
  vector[K-1] beta_raw;
}
transformed parameters{
   vector[K] beta =  A_qr * beta_raw;
}
model {
  beta_raw ~ normal(0, inv(sqrt(1 - inv(K))));
}
```

This produces a marginal standard normal distribution on the values of `beta`, which will sum to zero by construction of the QR decomposition.

Translated and Scaled Simplex

An alternative approach that's less efficient, but amenable to a symmetric prior, is to offset and scale a simplex.

```
parameters {
  simplex[K] beta_raw;
  real beta_scale;
```

```
  ...
transformed parameters {
  vector[K] beta;
  beta = beta_scale * (beta_raw - inv(K));
  ...
```

Here `inv(K)` is just a short way to write `1.0 / K`. Given that `beta_raw` sums to 1 because it is a simplex, the elementwise subtraction of `inv(K)` is guaranteed to sum to zero. Because the magnitude of the elements of the simplex is bounded, a scaling factor is required to provide `beta` with K degrees of freedom necessary to take on every possible value that sums to zero.

With this parameterization, a Dirichlet prior can be placed on `beta_raw`, perhaps uniform, and another prior put on `beta_scale`, typically for "shrinkage."

Soft Centering

Adding a prior such as $\beta \sim \mathsf{normal}(0, \sigma)$ will provide a kind of soft centering of a parameter vector β by preferring, all else being equal, that $\sum_{k=1}^{K} \beta_k = 0$. This approach is only guaranteed to roughly center if β and the elementwise addition $\beta + c$ for a scalar constant c produce the same likelihood (perhaps by another vector α being transformed to $\alpha - c$, as in the IRT models). This is another way of achieving a symmetric prior.

1.8. Ordered Logistic and Probit Regression

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

Ordered Logistic Regression

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

```
data {
  int<lower=2> K;
  int<lower=0> N;
  int<lower=1> D;
```

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

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

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

Ordered Probit

An ordered probit model could be coded in exactly the same way by swapping the cumulative logistic (`inv_logit`) for the cumulative normal (`Phi`).

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

```
    eta = x[n] * beta;
    theta[1] = 1 - Phi(eta - c[1]);
    for (k in 2:(K-1))
      theta[k] = Phi(eta - c[k-1]) - Phi(eta - c[k]);
    theta[K] = Phi(eta - c[K-1]);
    y[n] ~ categorical(theta);
  }
}
```

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

1.9. Hierarchical Logistic Regression

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

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

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

```
data {
  int<lower=1> D;
  int<lower=0> N;
  int<lower=1> L;
  int<lower=0,upper=1> y[N];
  int<lower=1,upper=L> ll[N];
```

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

The standard deviation parameter `sigma` gets an implicit uniform prior on $(0, \infty)$ because of its declaration with a lower-bound constraint of zero. Stan allows improper priors as long as the posterior is proper. Nevertheless, it is usually helpful to have informative or at least weakly informative priors for all parameters; see the regression priors section for recommendations on priors for regression coefficients and scales.

Optimizing the Model

Where possible, vectorizing sampling statements leads to faster log probability and derivative evaluations. The speed boost is not because loops are eliminated, but because vectorization allows sharing subcomputations in the log probability and gradient calculations and because it reduces the size of the expression tree required for gradient calculations.

The first optimization vectorizes the for-loop over `D` as

```
mu ~ normal(0, 100);
for (l in 1:L)
  beta[l] ~ normal(mu, sigma);
```

The declaration of `beta` as an array of vectors means that the expression `beta[l]` denotes a vector. Although `beta` could have been declared as a matrix, an array of vectors (or a two-dimensional array) is more efficient for accessing rows; see the

indexing efficiency section for more information on the efficiency tradeoffs among arrays, vectors, and matrices.

This model can be further sped up and at the same time made more arithmetically stable by replacing the application of inverse-logit inside the Bernoulli distribution with the logit-parameterized Bernoulli,[3]

```
for (n in 1:N)
  y[n] ~ bernoulli_logit(x[n] * beta[ll[n]]);
```

Unlike in R or BUGS, loops, array access and assignments are fast in Stan because they are translated directly to C++. In most cases, the cost of allocating and assigning to a container is more than made up for by the increased efficiency due to vectorizing the log probability and gradient calculations. Thus the following version is faster than the original formulation as a loop over a sampling statement.

```
{
  vector[N] x_beta_ll;
  for (n in 1:N)
    x_beta_ll[n] = x[n] * beta[ll[n]];
  y ~ bernoulli_logit(x_beta_ll);
}
```

The brackets introduce a new scope for the local variable `x_beta_ll`; alternatively, the variable may be declared at the top of the model block.

In some cases, such as the above, the local variable assignment leads to models that are less readable. The recommended practice in such cases is to first develop and debug the more transparent version of the model and only work on optimizations when the simpler formulation has been debugged.

1.10. Hierarchical Priors

Priors on priors, also known as "hyperpriors," should be treated the same way as priors on lower-level parameters in that as much prior information as is available should be brought to bear. Because hyperpriors often apply to only a handful of lower-level parameters, care must be taken to ensure the posterior is both proper and not overly sensitive either statistically or computationally to wide tails in the priors.

[3]The Bernoulli-logit distribution builds in the log link function, taking

$$\texttt{bernoulli\_logit}\left(y \mid \alpha\right) = \texttt{bernoulli}\left(y \mid \text{logit}^{-1}(\alpha)\right).$$

Boundary-Avoiding Priors for MLE in Hierarchical Models

The fundamental problem with maximum likelihood estimation (MLE) in the hierarchical model setting is that as the hierarchical variance drops and the values cluster around the hierarchical mean, the overall density grows without bound. As an illustration, consider a simple hierarchical linear regression (with fixed prior mean) of $y_n \in \mathbb{R}$ on $x_n \in \mathbb{R}^K$, formulated as

$$\begin{aligned} y_n &\sim \mathsf{normal}(x_n\beta, \sigma) \\ \beta_k &\sim \mathsf{normal}(0, \tau) \\ \tau &\sim \mathsf{Cauchy}(0, 2.5) \end{aligned}$$

In this case, as $\tau \to 0$ and $\beta_k \to 0$, the posterior density

$$p(\beta, \tau, \sigma | y, x) \propto p(y | x, \beta, \tau, \sigma)$$

grows without bound. See the plot of Neal's funnel density, which has similar behavior.

There is obviously no MLE estimate for β, τ, σ in such a case, and therefore the model must be modified if posterior modes are to be used for inference. The approach recommended by Chung et al. (2013) is to use a gamma distribution as a prior, such as

$$\sigma \sim \mathsf{Gamma}(2, 1/A),$$

for a reasonably large value of A, such as $A = 10$.

1.11. Item-Response Theory Models

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

Data Declaration with Missingness

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

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

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

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

1PL (Rasch) Model

The 1PL item-response model, also known as the Rasch model, has one parameter (1P) for questions and uses the logistic link function (L).

The model parameters are declared as follows.

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

The parameter `alpha[j]` is the ability coefficient for student `j` and `beta[k]` is the difficulty coefficient for question `k`. The non-standard parameterization used here also includes an intercept term `delta`, which represents the average student's response to the average question.[4]

The model itself is as follows.

```
model {
```

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

```
  alpha ~ std_normal();          // informative true prior
  beta ~ std_normal();           // informative true prior
  delta ~ normal(0.75, 1);       // informative true prior
  for (n in 1:N)
    y[n] ~ bernoulli_logit(alpha[jj[n]] - beta[kk[n]] + delta);
}
```

This model uses the logit-parameterized Bernoulli distribution, where

$$\texttt{bernoulli\_logit}\left(y \mid \alpha\right) = \texttt{bernoulli}\left(y \mid \text{logit}^{-1}(\alpha)\right).$$

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

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

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

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

Multilevel 2PL Model

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

```
parameters {
  real mu_beta;                 // mean question difficulty
  vector[J] alpha;              // ability for j - mean
  vector[K] beta;               // difficulty for k
  vector<lower=0>[K] gamma;     // discrimination of k
```

```
  real<lower=0> sigma_beta;    // scale of difficulties
  real<lower=0> sigma_gamma;   // scale of log discrimination
}
```

The parameters should be clearer after the model definition.

```
model {
  alpha ~ std_normal();
  beta ~ normal(0, sigma_beta);
  gamma ~ lognormal(0, sigma_gamma);
  mu_beta ~ cauchy(0, 5);
  sigma_beta ~ cauchy(0, 5);
  sigma_gamma ~ cauchy(0, 5);
  y ~ bernoulli_logit(gamma[kk] .* (alpha[jj] - (beta[kk] + mu_beta)));
}
```

The `std_normal` function is used here, defined by

$$\texttt{std\_normal}(y) = \mathsf{normal}\,(y \mid 0, 1)\,.$$

The sampling statement is also vectorized using elementwise multiplication; it is equivalent to

```
for (n in 1:N)
  y[n] ~ bernoulli_logit(gamma[kk[n]]
                         * (alpha[jj[n]] - (beta[kk[n]] + mu_beta));
```

The 2PL model is similar to the 1PL model, with the additional parameter `gamma[k]` modeling how discriminative question `k` is. If `gamma[k]` is greater than 1, responses are more attenuated with less chance of getting a question right at random. The parameter `gamma[k]` is constrained to be positive, which prohibits there being questions that are easier for students of lesser ability; such questions are not unheard of, but they tend to be eliminated from most testing situations where an IRT model would be applied.

The model is parameterized here with student abilities `alpha` being given a standard normal prior. This is to identify both the scale and the location of the parameters, both of which would be unidentified otherwise; see the problematic posteriors chapter for further discussion of identifiability. The difficulty and discrimination parameters `beta` and `gamma` then have varying scales given hierarchically in this model. They could also be given weakly informative non-hierarchical priors, such as

```
beta ~ normal(0, 5);
```

```
gamma ~ lognormal(0, 2);
```

The point is that the `alpha` determines the scale and location and `beta` and `gamma` are allowed to float.

The `beta` parameter is here given a non-centered parameterization, with parameter `mu_beta` serving as the mean `beta` location. An alternative would've been to take:

```
beta ~ normal(mu_beta, sigma_beta);
```

and

```
y[n] ~ bernoulli_logit(gamma[kk[n]] * (alpha[jj[n]] - beta[kk[n]]));
```

Non-centered parameterizations tend to be more efficient in hierarchical models; see the reparameterization section for more information on non-centered reparameterizations.

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

1.12. Priors for Identifiability

Location and Scale Invariance

One application of (hierarchical) priors is to identify the scale and/or location of a group of parameters. For example, in the IRT models discussed in the previous section, there is both a location and scale non-identifiability. With uniform priors, the posteriors will float in terms of both scale and location. See the collinearity section for a simple example of the problems this poses for estimation.

The non-identifiability is resolved by providing a standard normal (i.e., $\text{normal}(0, 1)$) prior on one group of coefficients, such as the student abilities. With a standard normal prior on the student abilities, the IRT model is identified in that the posterior will produce a group of estimates for student ability parameters that have a sample mean of close to zero and a sample variance of close to one. The difficulty and discrimination parameters for the questions should then be given a diffuse, or ideally a hierarchical prior, which will identify these parameters by scaling and locating relative to the student ability parameters.

Collinearity

Another case in which priors can help provide identifiability is in the case of collinearity in a linear regression. In linear regression, if two predictors are collinear (i.e, one is a linear function of the other), then their coefficients will have a correlation of 1 (or -1) in the posterior. This leads to non-identifiability. By placing normal priors on the coefficients, the maximum likelihood solution of two duplicated predictors (trivially collinear) will be half the value than would be obtained by only including one.

Separability

In a logistic regression, if a predictor is positive in cases of 1 outcomes and negative in cases of 0 outcomes, then the maximum likelihood estimate for the coefficient for that predictor diverges to infinity. This divergence can be controlled by providing a prior for the coefficient, which will "shrink" the estimate back toward zero and thus identify the model in the posterior.

Similar problems arise for sampling with improper flat priors. The sampler will try to draw large values. By providing a prior, the posterior will be concentrated around finite values, leading to well-behaved sampling.

1.13. Multivariate Priors for Hierarchical Models

In hierarchical regression models (and other situations), several individual-level variables may be assigned hierarchical priors. For example, a model with multiple varying intercepts and slopes within might assign them a multivariate prior.

As an example, the individuals might be people and the outcome income, with predictors such as education level and age, and the groups might be states or other geographic divisions. The effect of education level and age as well as an intercept might be allowed to vary by state. Furthermore, there might be state-level predictors, such as average state income and unemployment level.

Multivariate Regression Example

(Gelman and Hill 2007, Chapter 13, Chapter 17) includes a discussion of a hierarchical model with N individuals organized into J groups. Each individual has a predictor row vector x_n of size K; to unify the notation, they assume that $x_{n,1} = 1$ is a fixed "intercept" predictor. To encode group membership, they assume individual n belongs to group $jj[n] \in \{1, \ldots, J\}$. Each individual n also has an observed outcome y_n taking on real values.

Likelihood

The model is a linear regression with slope and intercept coefficients varying by group, so that β_j is the coefficient K-vector for group j. The likelihood function for individual n is then just

$$y_n \sim \text{normal}(x_n \, \beta_{jj[n]}, \, \sigma) \quad \text{for} \quad n \in \{1, \ldots, N\}.$$

Coefficient Prior

Gelman and Hill model the coefficient vectors β_j as being drawn from a multivariate distribution with mean vector μ and covariance matrix Σ,

$$\beta_j \sim \text{multivariate normal}(\mu, \, \Sigma) \quad \text{for} \quad j \in \{1, \ldots, J\}.$$

Below, we discuss the full model of Gelman and Hill, which uses group-level predictors to model μ; for now, we assume μ is a simple vector parameter.

Hyperpriors

For hierarchical modeling, the group-level mean vector μ and covariance matrix Σ must themselves be given priors. The group-level mean vector can be given a reasonable weakly-informative prior for independent coefficients, such as

$$\mu_j \sim \text{normal}(0, 5).$$

If more is known about the expected coefficient values $\beta_{j,k}$, this information can be incorporated into the prior for μ_k.

For the prior on the covariance matrix, Gelman and Hill suggest using a scaled inverse Wishart. That choice was motivated primarily by convenience as it is conjugate to the multivariate likelihood function and thus simplifies Gibbs sampling

In Stan, there is no restriction to conjugacy for multivariate priors, and we in fact recommend a slightly different approach. Like Gelman and Hill, we decompose our prior into a scale and a matrix, but are able to do so in a more natural way based on the actual variable scales and a correlation matrix. Specifically, we define

$$\Sigma = \texttt{diag\_matrix}(\tau) \times \Omega \times \texttt{diag\_matrix}(\tau),$$

where Ω is a correlation matrix and τ is the vector of coefficient scales. This mapping from scale vector τ and correlation matrix Ω can be inverted, using

$$\tau_k = \sqrt{\Sigma_{k,k}} \quad \text{and} \quad \Omega_{i,j} = \frac{\Sigma_{i,j}}{\tau_i \, \tau_j}.$$

The components of the scale vector τ can be given any reasonable prior for scales, but we recommend something weakly informative like a half-Cauchy distribution with a small scale, such as

$$\tau_k \sim \mathsf{Cauchy}(0, 2.5) \quad \text{for} \quad k \in \{1, \ldots, K\} \quad \text{constrained by} \quad \tau_k > 0.$$

As for the prior means, if there is information about the scale of variation of coefficients across groups, it should be incorporated into the prior for τ. For large numbers of exchangeable coefficients, the components of τ itself (perhaps excluding the intercept) may themselves be given a hierarchical prior.

Our final recommendation is to give the correlation matrix Ω an LKJ prior with shape $\eta \geq 1$,[5]

$$\Omega \sim \mathsf{LKJCorr}(\eta).$$

The LKJ correlation distribution is defined by

$$\mathsf{LKJCorr}\left(\Sigma \mid \eta\right) \propto \det\left(\Sigma\right)^{\eta - 1}.$$

The basic behavior of the LKJ correlation distribution is similar to that of a beta distribution. For $\eta = 1$, the result is a uniform distribution. Despite being the identity over correlation matrices, the marginal distribution over the entries in that matrix (i.e., the correlations) is not uniform between -1 and 1. Rather, it concentrates around zero as the dimensionality increases due to the complex constraints.

For $\eta > 1$, the density increasingly concentrates mass around the unit matrix, i.e., favoring less correlation. For $\eta < 1$, it increasingly concentrates mass in the other direction, i.e., favoring more correlation.

The LKJ prior may thus be used to control the expected amount of correlation among the parameters β_j. For a discussion of decomposing a covariance prior into a prior on correlation matrices and an independent prior on scales, see Barnard, McCulloch, and Meng (2000).

[5]The prior is named for Lewandowski, Kurowicka, and Joe, as it was derived by inverting the random correlation matrix generation strategy of Lewandowski, Kurowicka, and Joe (2009).

Group-Level Predictors for Prior Mean

To complete Gelman and Hill's model, suppose each group $j \in \{1, \ldots, J\}$ is supplied with an L-dimensional row-vector of group-level predictors u_j. The prior mean for the β_j can then itself be modeled as a regression, using an L-dimensional coefficient vector γ. The prior for the group-level coefficients then becomes

$$\beta_j \sim \text{multivariate normal}(u_j \, \gamma, \Sigma)$$

The group-level coefficients γ may themselves be given independent weakly informative priors, such as

$$\gamma_l \sim \text{normal}(0, 5).$$

As usual, information about the group-level means should be incorporated into this prior.

Coding the Model in Stan

The Stan code for the full hierarchical model with multivariate priors on the group-level coefficients and group-level prior means follows its definition.

```
data {
  int<lower=0> N;              // num individuals
  int<lower=1> K;              // num ind predictors
  int<lower=1> J;              // num groups
  int<lower=1> L;              // num group predictors
  int<lower=1,upper=J> jj[N];  // group for individual
  matrix[N, K] x;              // individual predictors
  row_vector[L] u[J];          // group predictors
  vector[N] y;                 // outcomes
}
parameters {
  corr_matrix[K] Omega;        // prior correlation
  vector<lower=0>[K] tau;      // prior scale
  matrix[L, K] gamma;          // group coeffs
  vector[K] beta[J];           // indiv coeffs by group
  real<lower=0> sigma;         // prediction error scale
}
model {
  tau ~ cauchy(0, 2.5);
```

```
  Omega ~ lkj_corr(2);
  to_vector(gamma) ~ normal(0, 5);
  {
    row_vector[K] u_gamma[J];
    for (j in 1:J)
      u_gamma[j] = u[j] * gamma;
    beta ~ multi_normal(u_gamma, quad_form_diag(Omega, tau));
  }
  for (n in 1:N)
    y[n] ~ normal(x[n] * beta[jj[n]], sigma);
}
```

The hyperprior covariance matrix is defined implicitly through the a quadratic form in the code because the correlation matrix `Omega` and scale vector `tau` are more natural to inspect in the output; to output `Sigma`, define it as a transformed parameter. The function `quad_form_diag` is defined so that `quad_form_diag(Sigma, tau)` is equivalent to `diag_matrix(tau) * Sigma * diag_matrix(tau)`, where `diag_matrix(tau)` returns the matrix with `tau` on the diagonal and zeroes off diagonal; the version using `quad_form_diag` should be faster. For details on these and other matrix arithmetic operators and functions, see the function reference manual.

Optimization through Vectorization

The code in the Stan program above can be sped up dramatically by replacing:

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

with the vectorized form:

```
{
  vector[N] x_beta_jj;
  for (n in 1:N)
    x_beta_jj[n] = x[n] * beta[jj[n]];
  y ~ normal(x_beta_jj, sigma);
}
```

The outer brackets create a local scope in which to define the variable `x_beta_jj`, which is then filled in a loop and used to define a vectorized sampling statement. The reason this is such a big win is that it allows us to take the log of sigma only

once and it greatly reduces the size of the resulting expression graph by packing all of the work into a single density function.

Although it is tempting to redeclare `beta` and include a revised model block sampling statement,

```
parameters {
  matrix[J, K] beta;
...
model {
  y ~ normal(rows_dot_product(x, beta[jj]), sigma);
  ...
```

this fails because it breaks the vectorization of sampling for `beta`,[6]

```
beta ~ multi_normal(...);
```

which requires `beta` to be an array of vectors. Both vectorizations are important, so the best solution is to just use the loop above, because `rows_dot_product` cannot do much optimization in and of itself because there are no shared computations.

The code in the Stan program above also builds up an array of vectors for the outcomes and for the multivariate normal, which provides a major speedup by reducing the number of linear systems that need to be solved and differentiated.

```
{
  matrix[K, K] Sigma_beta;
  Sigma_beta = quad_form_diag(Omega, tau);
  for (j in 1:J)
    beta[j] ~ multi_normal((u[j] * gamma)', Sigma_beta);
}
```

In this example, the covariance matrix `Sigma_beta` is defined as a local variable so as not to have to repeat the quadratic form computation J times. This vectorization can be combined with the Cholesky-factor optimization in the next section.

Optimization through Cholesky Factorization

The multivariate normal density and LKJ prior on correlation matrices both require their matrix parameters to be factored. Vectorizing, as in the previous section, ensures this is only done once for each density. An even better solution, both in

[6] Thanks to Mike Lawrence for pointing this out in the GitHub issue for the manual.

terms of efficiency and numerical stability, is to parameterize the model directly in terms of Cholesky factors of correlation matrices using the multivariate version of the non-centered parameterization. For the model in the previous section, the program fragment to replace the full matrix prior with an equivalent Cholesky factorized prior is as follows.

```
data {
  matrix[J, L] u;
  ...
parameters {
  matrix[K, J] z;
  cholesky_factor_corr[K] L_Omega;
  ...
transformed parameters {
  matrix[J, K] beta;
  beta = u * gamma + (diag_pre_multiply(tau,L_Omega) * z)';
}
model {
  to_vector(z) ~ std_normal();
  L_Omega ~ lkj_corr_cholesky(2);
  ...
```

The data variable `u` was originally an array of vectors, which is efficient for access; here it is redeclared as a matrix in order to use it in matrix arithmetic. The new parameter `L_Omega` is the Cholesky factor of the original correlation matrix `Omega`, so that

```
Omega = L_Omega * L_Omega'
```

The prior scale vector `tau` is unchanged, and furthermore, Pre-multiplying the Cholesky factor by the scale produces the Cholesky factor of the final covariance matrix,

```
Sigma_beta
  = quad_form_diag(Omega, tau)
  = diag_pre_multiply(tau, L_Omega) * diag_pre_multiply(tau, L_Omega)'
```

where the diagonal pre-multiply compound operation is defined by

```
diag_pre_multiply(a, b) = diag_matrix(a) * b
```

The new variable `z` is declared as a matrix, the entries of which are given independent standard normal priors; the `to_vector` operation turns the matrix into a vector so

that it can be used as a vectorized argument to the univariate normal density. Multiplying the Cholesky factor of the covariance matrix by `z` and adding the mean `(u * gamma)'` produces a `beta` distributed as in the original model.

Omitting the data declarations, which are the same as before, the optimized model is as follows.

```
parameters {
  matrix[K, J] z;
  cholesky_factor_corr[K] L_Omega;
  vector<lower=0,upper=pi()/2>[K] tau_unif;
  matrix[L, K] gamma;                          // group coeffs
  real<lower=0> sigma;                        // prediction error scale
}
transformed parameters {
  matrix[J, K] beta;
  vector<lower=0>[K] tau;     // prior scale
  for (k in 1:K) tau[k] = 2.5 * tan(tau_unif[k]);
  beta = u * gamma + (diag_pre_multiply(tau,L_Omega) * z)';
}
model {
  to_vector(z) ~ std_normal();
  L_Omega ~ lkj_corr_cholesky(2);
  to_vector(gamma) ~ normal(0, 5);
  y ~ normal(rows_dot_product(beta[jj] , x), sigma);
}
```

This model also reparameterizes the prior scale `tau` to avoid potential problems with the heavy tails of the Cauchy distribution. The statement `tau_unif ~ uniform(0,pi()/2)` can be omitted from the model block because Stan increments the log posterior for parameters with uniform priors without it.

1.14. Prediction, Forecasting, and Backcasting

Stan models can be used for "predicting" the values of arbitrary model unknowns. When predictions are about the future, they're called "forecasts;" when they are predictions about the past, as in climate reconstruction or cosmology, they are sometimes called "backcasts" (or "aftcasts" or "hindcasts" or "antecasts," depending on the author's feelings about the opposite of "fore").

Programming Predictions

As a simple example, the following linear regression provides the same setup for estimating the coefficients `beta` as in our first example above, using `y` for the `N` observations and `x` for the `N` predictor vectors. The model parameters and model for observations are exactly the same as before.

To make predictions, we need to be given the number of predictions, `N_new`, and their predictor matrix, `x_new`. The predictions themselves are modeled as a parameter `y_new`. The model statement for the predictions is exactly the same as for the observations, with the new outcome vector `y_new` and prediction matrix `x_new`.

```
data {
  int<lower=1> K;
  int<lower=0> N;
  matrix[N, K] x;
  vector[N] y;

  int<lower=0> N_new;
  matrix[N_new, K] x_new;
}
parameters {
  vector[K] beta;
  real<lower=0> sigma;

  vector[N_new] y_new;                    // predictions
}
model {
  y ~ normal(x * beta, sigma);            // observed model

  y_new ~ normal(x_new * beta, sigma);  // prediction model
}
```

Predictions as Generated Quantities

Where possible, the most efficient way to generate predictions is to use the generated quantities block. This provides proper Monte Carlo (not Markov chain Monte Carlo) inference, which can have a much higher effective sample size per iteration.

```
...data as above...

parameters {
```

```
  vector[K] beta;
  real<lower=0> sigma;
}
model {
  y ~ normal(x * beta, sigma);
}
generated quantities {
  vector[N_new] y_new;
  for (n in 1:N_new)
    y_new[n] = normal_rng(x_new[n] * beta, sigma);
}
```

Now the data are just as before, but the parameter `y_new` is now declared as a generated quantity, and the prediction model is removed from the model and replaced by a pseudo-random draw from a normal distribution.

Overflow in Generated Quantities

It is possible for values to overflow or underflow in generated quantities. The problem is that if the result is NaN, then any constraints placed on the variables will be violated. It is possible to check a value assigned by an RNG and reject it if it overflows, but this is both inefficient and leads to biased posterior estimates. Instead, the conditions causing overflow, such as trying to generate a negative binomial random variate with a mean of 2^{31}. These must be intercepted and dealt with, typically be reparameterizing or reimplementing the random number generator using real values rather than integers, which are upper-bounded by $2^{31} - 1$ in Stan.

1.15. Multivariate Outcomes

Most regressions are set up to model univariate observations (be they scalar, boolean, categorical, ordinal, or count). Even multinomial regressions are just repeated categorical regressions. In contrast, this section discusses regression when each observed value is multivariate. To relate multiple outcomes in a regression setting, their error terms are provided with covariance structure.

This section considers two cases, seemingly unrelated regressions for continuous multivariate quantities and multivariate probit regression for boolean multivariate quantities.

Seemingly Unrelated Regressions

The first model considered is the "seemingly unrelated" regressions (SUR) of econometrics where several linear regressions share predictors and use a covariance error structure rather than independent errors (Zellner 1962; Greene 2011).

The model is easy to write down as a regression,

$$
\begin{aligned}
y_n &= x_n\,\beta + \epsilon_n \\
\epsilon_n &\sim \text{multivariate normal}(0, \Sigma)
\end{aligned}
$$

where x_n is a J-row-vector of predictors (x is an $(N \times J)$ matrix), y_n is a K-vector of observations, β is a $(K \times J)$ matrix of regression coefficients (vector β_k holds coefficients for outcome k), and Σ is covariance matrix governing the error. As usual, the intercept can be rolled into x as a column of ones.

The basic Stan code is straightforward (though see below for more optimized code for use with LKJ priors on correlation).

```
data {
  int<lower=1> K;
  int<lower=1> J;
  int<lower=0> N;
  vector[J] x[N];
  vector[K] y[N];
}
parameters {
  matrix[K, J] beta;
  cov_matrix[K] Sigma;
}
model {
  vector[K] mu[N];
  for (n in 1:N)
    mu[n] = beta * x[n];
  y ~ multi_normal(mu, Sigma);
}
```

For efficiency, the multivariate normal is vectorized by precomputing the array of mean vectors and sharing the same covariance matrix.

Following the advice in the multivariate hierarchical priors section, we will place a weakly informative normal prior on the regression coefficients, an LKJ prior on the correlations and a half-Cauchy prior on standard deviations. The covariance structure is parameterized in terms of Cholesky factors for efficiency and arithmetic stability.

```
...
parameters {
  matrix[K, J] beta;
  cholesky_factor_corr[K] L_Omega;
  vector<lower=0>[K] L_sigma;
}
model {
  vector[K] mu[N];
  matrix[K, K] L_Sigma;

  for (n in 1:N)
    mu[n] = beta * x[n];

  L_Sigma = diag_pre_multiply(L_sigma, L_Omega);

  to_vector(beta) ~ normal(0, 5);
  L_Omega ~ lkj_corr_cholesky(4);
  L_sigma ~ cauchy(0, 2.5);

  y ~ multi_normal_cholesky(mu, L_Sigma);
}
```

The Cholesky factor of the covariance matrix is then reconstructed as a local variable and used in the model by scaling the Cholesky factor of the correlation matrices. The regression coefficients get a prior all at once by converting the matrix `beta` to a vector.

If required, the full correlation or covariance matrices may be reconstructed from their Cholesky factors in the generated quantities block.

Multivariate Probit Regression

The multivariate probit model generates sequences of boolean variables by applying a step function to the output of a seemingly unrelated regression.

The observations y_n are D-vectors of boolean values (coded 0 for false, 1 for true).

The values for the observations y_n are based on latent values z_n drawn from a seemingly unrelated regression model (see the previous section),

$$z_n = x_n \beta + \epsilon_n$$
$$\epsilon_n \sim \text{multivariate normal}(0, \Sigma)$$

These are then put through the step function to produce a K-vector z_n of boolean values with elements defined by

$$y_{n,k} = \mathrm{I}\left(z_{n,k} > 0\right),$$

where $\mathrm{I}()$ is the indicator function taking the value 1 if its argument is true and 0 otherwise.

Unlike in the seemingly unrelated regressions case, here the covariance matrix Σ has unit standard deviations (i.e., it is a correlation matrix). As with ordinary probit and logistic regressions, letting the scale vary causes the model (which is defined only by a cutpoint at 0, not a scale) to be unidentified (see Greene (2011)).

Multivariate probit regression can be coded in Stan using the trick introduced by Albert and Chib (1993), where the underlying continuous value vectors y_n are coded as truncated parameters. The key to coding the model in Stan is declaring the latent vector z in two parts, based on whether the corresponding value of y is 0 or 1. Otherwise, the model is identical to the seemingly unrelated regression model in the previous section.

First, we introduce a sum function for two-dimensional arrays of integers; this is going to help us calculate how many total 1 values there are in y.

```
functions {
  int sum2d(int[,] a) {
    int s = 0;
    for (i in 1:size(a))
      s += sum(a[i]);
    return s;
  }
}
```

The function is trivial, but it's not a built-in for Stan and it's easier to understand the rest of the model if it's pulled into its own function so as not to create a distraction.

The data declaration block is much like for the seemingly unrelated regressions, but the observations y are now integers constrained to be 0 or 1.

```
data {
  int<lower=1> K;
  int<lower=1> D;
  int<lower=0> N;
  int<lower=0,upper=1> y[N,D];
  vector[K] x[N];
}
```

After declaring the data, there is a rather involved transformed data block whose sole purpose is to sort the data array y into positive and negative components, keeping track of indexes so that z can be easily reassembled in the transformed parameters block.

```
transformed data {
  int<lower=0> N_pos;
  int<lower=1,upper=N> n_pos[sum2d(y)];
  int<lower=1,upper=D> d_pos[size(n_pos)];
  int<lower=0> N_neg;
  int<lower=1,upper=N> n_neg[(N * D) - size(n_pos)];
  int<lower=1,upper=D> d_neg[size(n_neg)];

  N_pos = size(n_pos);
  N_neg = size(n_neg);
  {
    int i;
    int j;
    i = 1;
    j = 1;
    for (n in 1:N) {
      for (d in 1:D) {
        if (y[n,d] == 1) {
          n_pos[i] = n;
          d_pos[i] = d;
          i += 1;
        } else {
          n_neg[j] = n;
          d_neg[j] = d;
          j += 1;
```

```
        }
      }
    }
  }
}
```

The variables `N_pos` and `N_neg` are set to the number of true (1) and number of false (0) observations in `y`. The loop then fills in the sequence of indexes for the positive and negative values in four arrays.

The parameters are declared as follows.

```
parameters {
  matrix[D, K] beta;
  cholesky_factor_corr[D] L_Omega;
  vector<lower=0>[N_pos] z_pos;
  vector<upper=0>[N_neg] z_neg;
}
```

These include the regression coefficients `beta` and the Cholesky factor of the correlation matrix, `L_Omega`. This time there is no scaling because the covariance matrix has unit scale (i.e., it is a correlation matrix; see above).

The critical part of the parameter declaration is that the latent real value z is broken into positive-constrained and negative-constrained components, whose size was conveniently calculated in the transformed data block. The transformed data block's real work was to allow the transformed parameter block to reconstruct z.

```
transformed parameters {
  vector[D] z[N];
  for (n in 1:N_pos)
    z[n_pos[n], d_pos[n]] = z_pos[n];
  for (n in 1:N_neg)
    z[n_neg[n], d_neg[n]] = z_neg[n];
}
```

At this point, the model is simple, pretty much recreating the seemingly unrelated regression.

```
model {
  L_Omega ~ lkj_corr_cholesky(4);
  to_vector(beta) ~ normal(0, 5);
  {
```

```
    vector[D] beta_x[N];
    for (n in 1:N)
      beta_x[n] = beta * x[n];
    z ~ multi_normal_cholesky(beta_x, L_Omega);
  }
}
```

This simple form of model is made possible by the Albert and Chib-style constraints on z.

Finally, the correlation matrix itself can be put back together in the generated quantities block if desired.

```
generated quantities {
  corr_matrix[D] Omega;
  Omega = multiply_lower_tri_self_transpose(L_Omega);
}
```

The same could be done for the seemingly unrelated regressions in the previous section.

1.16. Applications of Pseudorandom Number Generation

The main application of pseudorandom number generator (PRNGs) is for posterior inference, including prediction and posterior predictive checks. They can also be used for pure data simulation, which is like a posterior predictive check with no conditioning. See the function reference manual for a complete description of the syntax and usage of pseudorandom number generators.

Prediction

Consider predicting unobserved outcomes using linear regression. Given predictors $x_1, \ldots, x_N$ and observed outcomes $y_1, \ldots, y_N$, and assuming a standard linear regression with intercept α, slope β, and error scale σ, along with improper uniform priors, the posterior over the parameters given x and y is

$$p\left(\alpha, \beta, \sigma \mid x, y\right) \propto \prod_{n=1}^{N} \mathsf{normal}\left(y_n \mid \alpha + \beta x_n, \sigma\right).$$

For this model, the posterior predictive inference for a new outcome $\tilde{y}_m$ given a

predictor $\tilde{x}_m$, conditioned on the observed data x and y, is

$$p\left(\tilde{y}_n \mid \tilde{x}_n, x, y\right) = \int_{(\alpha,\beta,\sigma)} \mathsf{normal}\left(\tilde{y}_n \mid \alpha + \beta\tilde{x}_n, \sigma\right) \times p\left(\alpha, \beta, \sigma \mid x, y\right) \, \mathrm{d}(\alpha, \beta, \sigma).$$

To code the posterior predictive inference in Stan, a standard linear regression is combined with a random number in the generated quantities block.

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

Given observed predictors x and outcomes y, `y_tilde` will be drawn according to $p\left(\tilde{y} \mid \tilde{x}, y, x\right)$. This means that, for example, the posterior mean for `y_tilde` is the estimate of the outcome that minimizes expected square error (conditioned on the data and model).

Posterior Predictive Checks

A good way to investigate the fit of a model to the data, a critical step in Bayesian data analysis, is to generate simulated data according to the parameters of the model. This is carried out with exactly the same procedure as before, only the observed data predictors x are used in place of new predictors $\tilde{x}$ for unobserved outcomes. If the model fits the data well, the predictions for $\tilde{y}$ based on x should match the observed data y.

To code posterior predictive checks in Stan requires only a slight modification of the prediction code to use x and N in place of $\tilde{x}$ and $\tilde{N}$,

```
generated quantities {
  vector[N] y_tilde;
  for (n in 1:N)
    y_tilde[n] = normal_rng(alpha + beta * x[n], sigma);
}
```

Gelman et al. (2013) recommend choosing several posterior draws $\tilde{y}^{(1)}, \ldots, \tilde{y}^{(M)}$ and plotting each of them alongside the data y that was actually observed. If the model fits well, the simulated $\tilde{y}$ will look like the actual data y.

2. Time-Series Models

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

The Gaussian processes chapter presents Gaussian processes, which may also be used for time-series (and spatial) data.

2.1. Autoregressive Models

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

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

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

AR(1) Models

With improper flat priors on the regression coefficients α and β and on the positively-constrained noise scale (σ), the Stan program for the AR(1) model is as follows.[1]

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

[1]The intercept in this model is $\alpha/(1-\beta)$. An alternative parameterization in terms of an intercept γ suggested Mark Scheuerell on GitHub is $y_n \sim \mathsf{normal}\,(\gamma + \beta \cdot (y_{n-1} - \gamma), \sigma)$.

The first observed data point, `y[1]`, is not modeled here because there is nothing to condition on; instead, it acts to condition `y[2]`. This model also uses an improper prior for `sigma`, but there is no obstacle to adding an informative prior if information is available on the scale of the changes in `y` over time, or a weakly informative prior to help guide inference if rough knowledge of the scale of `y` is available.

Slicing for Efficiency

Although perhaps a bit more difficult to read, a much more efficient way to write the above model is by slicing the vectors, with the model above being replaced with the one-liner

```
model {
  y[2:N] ~ normal(alpha + beta * y[1:(N - 1)], sigma);
}
```

The left-hand side slicing operation pulls out the last $N - 1$ elements and the right-hand side version pulls out the first $N - 1$.

Extensions to the AR(1) Model

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

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

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

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

AR(2) Models

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

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

AR(K) Models

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

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

ARCH(1) Models

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

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

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

In order to ensure the noise terms σ_t^2 are positive, the scale coefficients are constrained to be positive, $\alpha_0, \alpha_1 > 0$. To ensure stationarity of the time series, the slope is constrained to to be less than one, i.e., $\alpha_1 < 1$.[2]

The ARCH(1) model may be coded directly in Stan as follows.

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

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

2.2. Modeling Temporal Heteroscedasticity

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

[2] In practice, it can be useful to remove the constraint to test whether a non-stationary set of coefficients provides a better fit to the data. It can also be useful to add a trend term to the model, because an unfitted trend will manifest as non-stationarity.

GARCH(1,1) Models

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

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

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

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

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

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

A transformed parameter array of non-negative values `sigma` is used to store the scale values at each time point. The definition of these values in the transformed

parameters block is where the regression is now defined. There is an intercept `alpha0`, a slope `alpha1` for the squared difference in return from the mean at the previous time, and a slope `beta1` for the previous noise scale squared. Finally, the whole regression is inside the `sqrt` function because Stan requires scale (deviation) parameters (not variance parameters) for the normal distribution.

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

2.3. Moving Average Models

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

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

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

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

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

MA(2) Example

An MA(2) model can be coded in Stan as follows.

```
data {
  int<lower=3> T;           // number of observations
  vector[T] y;              // observation at time T
}
parameters {
  real mu;                  // mean
  real<lower=0> sigma;      // error scale
  vector[2] theta;          // lag coefficients
}
transformed parameters {
  vector[T] epsilon;        // error terms
  epsilon[1] = y[1] - mu;
  epsilon[2] = y[2] - mu - theta[1] * epsilon[1];
```

```
  for (t in 3:T)
    epsilon[t] = ( y[t] - mu
                   - theta[1] * epsilon[t - 1]
                   - theta[2] * epsilon[t - 2] );
}
model {
  mu ~ cauchy(0, 2.5);
  theta ~ cauchy(0, 2.5);
  sigma ~ cauchy(0, 2.5);
  for (t in 3:T)
    y[t] ~ normal(mu
                  + theta[1] * epsilon[t - 1]
                  + theta[2] * epsilon[t - 2],
                  sigma);
}
```

The error terms ϵ_t are defined as transformed parameters in terms of the observations and parameters. The definition of the sampling statement (defining the likelihood) follows the definition, which can only be applied to y_n for $n > Q$. In this example, the parameters are all given Cauchy (half-Cauchy for σ) priors, although other priors can be used just as easily.

This model could be improved in terms of speed by vectorizing the sampling statement in the model block. Vectorizing the calculation of the ϵ_t could also be sped up by using a dot product instead of a loop.

Vectorized MA(Q) Model

A general MA(Q) model with a vectorized sampling probability may be defined as follows.

```
data {
  int<lower=0> Q;       // num previous noise terms
  int<lower=3> T;       // num observations
  vector[T] y;          // observation at time t
}
parameters {
  real mu;              // mean
  real<lower=0> sigma;  // error scale
  vector[Q] theta;      // error coeff, lag -t
}
```

```
transformed parameters {
  vector[T] epsilon;    // error term at time t
  for (t in 1:T) {
    epsilon[t] = y[t] - mu;
    for (q in 1:min(t - 1, Q))
      epsilon[t] = epsilon[t] - theta[q] * epsilon[t - q];
  }
}
model {
  vector[T] eta;
  mu ~ cauchy(0, 2.5);
  theta ~ cauchy(0, 2.5);
  sigma ~ cauchy(0, 2.5);
  for (t in 1:T) {
    eta[t] = mu;
    for (q in 1:min(t - 1, Q))
      eta[t] = eta[t] + theta[q] * epsilon[t - q];
  }
  y ~ normal(eta, sigma);
}
```

Here all of the data are modeled, with missing terms just dropped from the regressions as in the calculation of the error terms. Both models converge quickly and mix well at convergence, with the vectorized model being faster (per iteration, not to converge—they compute the same model).

2.4. Autoregressive Moving Average Models

Autoregressive moving-average models (ARMA), combine the predictors of the autoregressive model and the moving average model. An ARMA(1,1) model, with a single state of history, can be encoded in Stan as follows.

```
data {
  int<lower=1> T;             // num observations
  real y[T];                  // observed outputs
}
parameters {
  real mu;                    // mean coeff
  real phi;                   // autoregression coeff
  real theta;                 // moving avg coeff
  real<lower=0> sigma;        // noise scale
```

```
}
model {
  vector[T] nu;               // prediction for time t
  vector[T] err;              // error for time t
  nu[1] = mu + phi * mu;      // assume err[0] == 0
  err[1] = y[1] - nu[1];
  for (t in 2:T) {
    nu[t] = mu + phi * y[t-1] + theta * err[t-1];
    err[t] = y[t] - nu[t];
  }
  mu ~ normal(0, 10);         // priors
  phi ~ normal(0, 2);
  theta ~ normal(0, 2);
  sigma ~ cauchy(0, 5);
  err ~ normal(0, sigma);     // likelihood
}
```

The data are declared in the same way as the other time-series regressions and the parameters are documented in the code.

In the model block, the local vector `nu` stores the predictions and `err` the errors. These are computed similarly to the errors in the moving average models described in the previous section.

The priors are weakly informative for stationary processes. The likelihood only involves the error term, which is efficiently vectorized here.

Often in models such as these, it is desirable to inspect the calculated error terms. This could easily be accomplished in Stan by declaring `err` as a transformed parameter, then defining it the same way as in the model above. The vector `nu` could still be a local variable, only now it will be in the transformed parameter block.

Wayne Folta suggested encoding the model without local vector variables as follows.

```
model {
  real err;
  mu ~ normal(0, 10);
  phi ~ normal(0, 2);
  theta ~ normal(0, 2);
  sigma ~ cauchy(0, 5);
  err = y[1] - mu + phi * mu;
  err ~ normal(0, sigma);
```

```
  for (t in 2:T) {
    err = y[t] - (mu + phi * y[t-1] + theta * err);
    err ~ normal(0, sigma);
  }
}
```

This approach to ARMA models illustrates how local variables, such as `err` in this case, can be reused in Stan. Folta's approach could be extended to higher order moving-average models by storing more than one error term as a local variable and reassigning them in the loop.

Both encodings are fast. The original encoding has the advantage of vectorizing the normal distribution, but it uses a bit more memory. A halfway point would be to vectorize just `err`.

Identifiability and Stationarity

MA and ARMA models are not identifiable if the roots of the characteristic polynomial for the MA part lie inside the unit circle, so it's necessary to add the following constraint[3]

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

When the model is run without the constraint, using synthetic data generated from the model, the simulation can sometimes find modes for (`theta`, `phi`) outside the $[-1, 1]$ interval, which creates a multiple mode problem in the posterior and also causes the NUTS tree depth to get large (often above 10). Adding the constraint both improves the accuracy of the posterior and dramatically reduces the tree depth, which speeds up the simulation considerably (typically by much more than an order of magnitude).

Further, unless one thinks that the process is really non-stationary, it's worth adding the following constraint to ensure stationarity.

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

2.5. Stochastic Volatility Models

Stochastic volatility models treat the volatility (i.e., variance) of a return on an asset, such as an option to buy a security, as following a latent stochastic process in discrete time Kim, Shephard, and Chib (1998). The data consist of mean corrected (i.e.,

[3]This subsection is a lightly edited comment of Jonathan Gilligan's on GitHub; see https://github.com/stan-dev/stan/issues/1617#issuecomment-160249142

centered) returns y_t on an underlying asset at T equally spaced time points. Kim et al. formulate a typical stochastic volatility model using the following regression-like equations, with a latent parameter h_t for the log volatility, along with parameters μ for the mean log volatility, and ϕ for the persistence of the volatility term. The variable ϵ_t represents the white-noise shock (i.e., multiplicative error) on the asset return at time t, whereas δ_t represents the shock on volatility at time t.

$$
\begin{aligned}
y_t &= \epsilon_t \exp(h_t/2) \\
h_{t+1} &= \mu + \phi(h_t - \mu) + \delta_t \sigma \\
h_1 &\sim \mathsf{normal}\left(\mu, \frac{\sigma}{\sqrt{1-\phi^2}}\right) \\
\epsilon_t &\sim \mathsf{normal}(0, 1) \\
\delta_t &\sim \mathsf{normal}(0, 1)
\end{aligned}
$$

Rearranging the first line, $\epsilon_t = y_t \exp(-h_t/2)$, allowing the sampling distribution for y_t to be written as

$$y_t \sim \mathsf{normal}(0, \exp(h_t/2)).$$

The recurrence equation for h_{t+1} may be combined with the scaling and sampling of δ_t to yield the sampling distribution

$$h_t \sim \mathsf{normal}(\mu + \phi(h_{t-1} - \mu), \sigma).$$

This formulation can be directly encoded, as shown in the following Stan model.

```
data {
  int<lower=0> T;   // # time points (equally spaced)
  vector[T] y;      // mean corrected return at time t
}
parameters {
  real mu;                     // mean log volatility
  real<lower=-1,upper=1> phi;  // persistence of volatility
  real<lower=0> sigma;         // white noise shock scale
  vector[T] h;                 // log volatility at time t
}
model {
  phi ~ uniform(-1, 1);
  sigma ~ cauchy(0, 5);
```

```
  mu ~ cauchy(0, 10);
  h[1] ~ normal(mu, sigma / sqrt(1 - phi * phi));
  for (t in 2:T)
    h[t] ~ normal(mu + phi * (h[t - 1] -  mu), sigma);
  for (t in 1:T)
    y[t] ~ normal(0, exp(h[t] / 2));
}
```

Compared to the Kim et al. formulation, the Stan model adds priors for the parameters ϕ, σ, and μ. The shock terms ϵ_t and δ_t do not appear explicitly in the model, although they could be calculated efficiently in a generated quantities block.

The posterior of a stochastic volatility model such as this one typically has high posterior variance. For example, simulating 500 data points from the above model with $\mu = -1.02$, $\phi = 0.95$, and $\sigma = 0.25$ leads to 95% posterior intervals for μ of $(-1.23, -0.54)$, for ϕ of $(0.82, 0.98)$, and for σ of $(0.16, 0.38)$.

The samples using NUTS show a high degree of autocorrelation among the samples, both for this model and the stochastic volatility model evaluated in (Hoffman and Gelman 2011; Hoffman and Gelman 2014). Using a non-diagonal mass matrix provides faster convergence and more effective samples than a diagonal mass matrix, but will not scale to large values of T.

It is relatively straightforward to speed up the effective samples per second generated by this model by one or more orders of magnitude. First, the sampling statements for return y is easily vectorized to

```
y ~ normal(0, exp(h / 2));
```

This speeds up the iterations, but does not change the effective sample size because the underlying parameterization and log probability function have not changed. Mixing is improved by by reparameterizing in terms of a standardized volatility, then rescaling. This requires a standardized parameter `h_std` to be declared instead of `h`.

```
parameters {
  ...
  vector[T] h_std;  // std log volatility time t
```

The original value of `h` is then defined in a transformed parameter block.

```
transformed parameters {
  vector[T] h = h_std * sigma;  // now h ~ normal(0, sigma)
  h[1] /= sqrt(1 - phi * phi);  // rescale h[1]
  h += mu;
```

```
  for (t in 2:T)
    h[t] += phi * (h[t-1] - mu);
}
```

The first assignment rescales `h_std` to have a $\mathsf{normal}(0, \sigma)$ distribution and temporarily assigns it to `h`. The second assignment rescales `h[1]` so that its prior differs from that of `h[2]` through `h[T]`. The next assignment supplies a `mu` offset, so that `h[2]` through `h[T]` are now distributed $\mathsf{normal}(\mu, \sigma)$; note that this shift must be done after the rescaling of `h[1]`. The final loop adds in the moving average so that `h[2]` through `h[T]` are appropriately modeled relative to `phi` and `mu`.

As a final improvement, the sampling statement for `h[1]` and loop for sampling `h[2]` to `h[T]` are replaced with a single vectorized standard normal sampling statement.

```
model {
  ...
  h_std ~ std_normal();
```

Although the original model can take hundreds and sometimes thousands of iterations to converge, the reparameterized model reliably converges in tens of iterations. Mixing is also dramatically improved, which results in higher effective sample sizes per iteration. Finally, each iteration runs in roughly a quarter of the time of the original iterations.

2.6. Hidden Markov Models

A hidden Markov model (HMM) generates a sequence of T output variables y_t conditioned on a parallel sequence of latent categorical state variables $z_t \in \{1, \ldots, K\}$. These "hidden" state variables are assumed to form a Markov chain so that z_t is conditionally independent of other variables given z_{t-1}. This Markov chain is parameterized by a transition matrix θ where θ_k is a K-simplex for $k \in \{1, \ldots, K\}$. The probability of transitioning to state z_t from state z_{t-1} is

$$z_t \sim \mathsf{categorical}(\theta_{z[t-1]}).$$

The output y_t at time t is generated conditionally independently based on the latent state z_t.

This section describes HMMs with a simple categorical model for outputs $y_t \in \{1, \ldots, V\}$. The categorical distribution for latent state k is parameterized by a V-simplex ϕ_k. The observed output y_t at time t is generated based on the hidden state indicator z_t at time t,

$$y_t \sim \mathsf{categorical}(\phi_{z[t]}).$$

In short, HMMs form a discrete mixture model where the mixture component indicators form a latent Markov chain.

Supervised Parameter Estimation

In the situation where the hidden states are known, the following naive model can be used to fit the parameters θ and ϕ.

```
data {
  int<lower=1> K;              // num categories
  int<lower=1> V;              // num words
  int<lower=0> T;              // num instances
  int<lower=1,upper=V> w[T];   // words
  int<lower=1,upper=K> z[T];   // categories
  vector<lower=0>[K] alpha;    // transit prior
  vector<lower=0>[V] beta;     // emit prior
}
parameters {
  simplex[K] theta[K];         // transit probs
  simplex[V] phi[K];           // emit probs
}
model {
  for (k in 1:K)
    theta[k] ~ dirichlet(alpha);
  for (k in 1:K)
    phi[k] ~ dirichlet(beta);
  for (t in 1:T)
    w[t] ~ categorical(phi[z[t]]);
  for (t in 2:T)
    z[t] ~ categorical(theta[z[t - 1]]);
}
```

Explicit Dirichlet priors have been provided for θ_k and ϕ_k; dropping these two statements would implicitly take the prior to be uniform over all valid simplexes.

Start-State and End-State Probabilities

Although workable, the above description of HMMs is incomplete because the start state z_1 is not modeled (the index runs from 2 to T). If the data are conceived as a subsequence of a long-running process, the probability of z_1 should be set to the stationary state probabilities in the Markov chain. In this case, there is no distinct

end to the data, so there is no need to model the probability that the sequence ends at z_T.

An alternative conception of HMMs is as models of finite-length sequences. For example, human language sentences have distinct starting distributions (usually a capital letter) and ending distributions (usually some kind of punctuation). The simplest way to model the sequence boundaries is to add a new latent state $K + 1$, generate the first state from a categorical distribution with parameter vector θ_{K+1}, and restrict the transitions so that a transition to state $K + 1$ is forced to occur at the end of the sentence and is prohibited elsewhere.

Calculating Sufficient Statistics

The naive HMM estimation model presented above can be sped up dramatically by replacing the loops over categorical distributions with a single multinomial distribution.

The data are declared as before. The transformed data block computes the sufficient statistics for estimating the transition and emission matrices.

```
transformed data {
  int<lower=0> trans[K, K];
  int<lower=0> emit[K, V];
  for (k1 in 1:K)
    for (k2 in 1:K)
      trans[k1, k2] = 0;
  for (t in 2:T)
    trans[z[t - 1], z[t]] += 1;
  for (k in 1:K)
    for (v in 1:V)
      emit[k,v] = 0;
  for (t in 1:T)
    emit[z[t], w[t]] += 1;
}
```

The likelihood component of the model based on looping over the input is replaced with multinomials as follows.

```
model {
  ...
  for (k in 1:K)
    trans[k] ~ multinomial(theta[k]);
```

```
  for (k in 1:K)
    emit[k] ~ multinomial(phi[k]);
}
```

In a continuous HMM with normal emission probabilities could be sped up in the same way by computing sufficient statistics.

Analytic Posterior

With the Dirichlet-multinomial HMM, the posterior can be computed analytically because the Dirichlet is the conjugate prior to the multinomial. The following example illustrates how a Stan model can define the posterior analytically. This is possible in the Stan language because the model only needs to define the conditional probability of the parameters given the data up to a proportion, which can be done by defining the (unnormalized) joint probability or the (unnormalized) conditional posterior, or anything in between.

The model has the same data and parameters as the previous models, but now computes the posterior Dirichlet parameters in the transformed data block.

```
transformed data {
  vector<lower=0>[K] alpha_post[K];
  vector<lower=0>[V] beta_post[K];
  for (k in 1:K)
    alpha_post[k] = alpha;
  for (t in 2:T)
    alpha_post[z[t-1], z[t]] += 1;
  for (k in 1:K)
    beta_post[k] = beta;
  for (t in 1:T)
    beta_post[z[t], w[t]] += 1;
}
```

The posterior can now be written analytically as follows.

```
model {
  for (k in 1:K)
    theta[k] ~ dirichlet(alpha_post[k]);
  for (k in 1:K)
    phi[k] ~ dirichlet(beta_post[k]);
}
```

Semisupervised Estimation

HMMs can be estimated in a fully unsupervised fashion without any data for which latent states are known. The resulting posteriors are typically extremely multimodal. An intermediate solution is to use semisupervised estimation, which is based on a combination of supervised and unsupervised data. Implementing this estimation strategy in Stan requires calculating the probability of an output sequence with an unknown state sequence. This is a marginalization problem, and for HMMs, it is computed with the so-called forward algorithm.

In Stan, the forward algorithm is coded as follows. First, two additional data variable are declared for the unsupervised data.

```
data {
  ...
  int<lower=1> T_unsup;              // num unsupervised items
  int<lower=1,upper=V> u[T_unsup];  // unsup words
  ...
```

The model for the supervised data does not change; the unsupervised data are handled with the following Stan implementation of the forward algorithm.

```
model {
 ...
  {
    real acc[K];
    real gamma[T_unsup, K];
    for (k in 1:K)
      gamma[1, k] = log(phi[k, u[1]]);
    for (t in 2:T_unsup) {
      for (k in 1:K) {
        for (j in 1:K)
          acc[j] = gamma[t-1, j] + log(theta[j, k]) + log(phi[k, u[t]]);
        gamma[t, k] = log_sum_exp(acc);
      }
    }
    target += log_sum_exp(gamma[T_unsup]);
  }
```

The forward values `gamma[t, k]` are defined to be the log marginal probability of the inputs `u[1],...,u[t]` up to time `t` and the latent state being equal to `k` at time `t`; the previous latent states are marginalized out. The first row of `gamma`

is initialized by setting `gamma[1, k]` equal to the log probability of latent state `k` generating the first output `u[1]`; as before, the probability of the first latent state is not itself modeled. For each subsequent time `t` and output `j`, the value `acc[j]` is set to the probability of the latent state at time `t-1` being `j`, plus the log transition probability from state `j` at time `t-1` to state `k` at time `t`, plus the log probability of the output `u[t]` being generated by state `k`. The `log_sum_exp` operation just multiplies the probabilities for each prior state `j` on the log scale in an arithmetically stable way.

The brackets provide the scope for the local variables `acc` and `gamma`; these could have been declared earlier, but it is clearer to keep their declaration near their use.

Predictive Inference

Given the transition and emission parameters, $\theta_{k,k'}$ and $\phi_{k,v}$ and an observation sequence $u_1, \ldots, u_T \in \{1, \ldots, V\}$, the Viterbi (dynamic programming) algorithm computes the state sequence which is most likely to have generated the observed output u.

The Viterbi algorithm can be coded in Stan in the generated quantities block as follows. The predictions here is the most likely state sequence `y_star[1], ..., y_star[T_unsup]` underlying the array of observations `u[1], ..., u[T_unsup]`. Because this sequence is determined from the transition probabilities `theta` and emission probabilities `phi`, it may be different from sample to sample in the posterior.

```
generated quantities {
  int<lower=1,upper=K> y_star[T_unsup];
  real log_p_y_star;
  {
    int back_ptr[T_unsup, K];
    real best_logp[T_unsup, K];
    real best_total_logp;
    for (k in 1:K)
      best_logp[1, k] = log(phi[k, u[1]]);
    for (t in 2:T_unsup) {
      for (k in 1:K) {
        best_logp[t, k] = negative_infinity();
        for (j in 1:K) {
          real logp;
          logp = best_logp[t-1, j]
                  + log(theta[j, k]) + log(phi[k, u[t]]);
```

```
          if (logp > best_logp[t, k]) {
            back_ptr[t, k] = j;
            best_logp[t, k] = logp;
          }
        }
      }
    }
    log_p_y_star = max(best_logp[T_unsup]);
    for (k in 1:K)
      if (best_logp[T_unsup, k] == log_p_y_star)
        y_star[T_unsup] = k;
    for (t in 1:(T_unsup - 1))
      y_star[T_unsup - t] = back_ptr[T_unsup - t + 1,
                                     y_star[T_unsup - t + 1]];
  }
}
```

The bracketed block is used to make the three variables `back_ptr`, `best_logp`, and `best_total_logp` local so they will not be output. The variable `y_star` will hold the label sequence with the highest probability given the input sequence `u`. Unlike the forward algorithm, where the intermediate quantities were total probability, here they consist of the maximum probability `best_logp[t, k]` for the sequence up to time `t` with final output category `k` for time `t`, along with a backpointer to the source of the link. Following the backpointers from the best final log probability for the final time `t` yields the optimal state sequence.

This inference can be run for the same unsupervised outputs `u` as are used to fit the semisupervised model. The above code can be found in the same model file as the unsupervised fit. This is the Bayesian approach to inference, where the data being reasoned about is used in a semisupervised way to train the model. It is not "cheating" because the underlying states for `u` are never observed — they are just estimated along with all of the other parameters.

If the outputs `u` are not used for semisupervised estimation but simply as the basis for prediction, the result is equivalent to what is represented in the BUGS modeling language via the cut operation. That is, the model is fit independently of `u`, then those parameters used to find the most likely state to have generated `u`.

3. Missing Data and Partially Known Parameters

Bayesian inference supports a general approach to missing data in which any missing data item is represented as a parameter that is estimated in the posterior Gelman et al. (2013). If the missing data are not explicitly modeled, as in the predictors for most regression models, then the result is an improper prior on the parameter representing the missing predictor.

Mixing arrays of observed and missing data can be difficult to include in Stan, partly because it can be tricky to model discrete unknowns in Stan and partly because unlike some other statistical languages (for example, R and Bugs), Stan requires observed and unknown quantities to be defined in separate places in the model. Thus it can be necessary to include code in a Stan program to splice together observed and missing parts of a data structure. Examples are provided later in the chapter.

3.1. Missing Data

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

An example involving missing normal observation could be coded as follows.[1]

```
data {
  int<lower=0> N_obs;
  int<lower=0> N_mis;
  real y_obs[N_obs];
}
parameters {
  real mu;
  real<lower=0> sigma;
  real y_mis[N_mis];
}
model {
  y_obs ~ normal(mu, sigma);
```

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

```
  y_mis ~ normal(mu, sigma);
}
```

The number of observed and missing data points are coded as data with non-negative integer variables `N_obs` and `N_mis`. The observed data are provided as an array data variable `y_obs`. The missing data are coded as an array parameter, `y_mis`. The ordinary parameters being estimated, the location `mu` and scale `sigma`, are also coded as parameters. The model is vectorized on the observed and missing data; combining them in this case would be less efficient because the data observations would be promoted and have needless derivatives calculated.

3.2. Partially Known Parameters

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

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

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

```
}
```

The variances are defined as data in variables `var1` and `var2`, whereas the covariance is defined as a parameter in variable `cov`. The 2×2 covariance matrix `Sigma` is defined as a transformed parameter, with the variances assigned to the two diagonal elements and the covariance to the two off-diagonal elements.

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

The vectorization of the multivariate normal is critical for efficiency here. The transformed parameter `Sigma` could be defined as a local variable within the model block if it does not need to be included in the sampler's output.

3.3. Sliced Missing Data

If the missing data are part of some larger data structure, then it can often be effectively reassembled using index arrays and slicing. Here's an example for time-series data, where only some entries in the series are observed.

```
data {
  int<lower = 0> N_obs;
  int<lower = 0> N_mis;
  int<lower = 1, upper = N_obs + N_mis> ii_obs[N_obs];
  int<lower = 1, upper = N_obs + N_mis> ii_mis[N_mis];
  real y_obs[N_obs];
}
transformed data {
  int<lower = 0> N = N_obs + N_mis;
}
parameters {
  real y_mis[N_mis];
  real<lower=0> sigma;
}
transformed parameters {
  real y[N];
  y[ii_obs] = y_obs;
  y[ii_mis] = y_mis;
}
model {
```

```
  sigma ~ gamma(1, 1);
  y[1] ~ normal(0, 100);
  y[2:N] ~ normal(y[1:(N - 1)], sigma);
}
```

The index arrays `ii_obs` and `ii_mis` contain the indexes into the final array `y` of the observed data (coded as a data vector `y_obs`) and the missing data (coded as a parameter vector `y_mis`). See the time series chapter for further discussion of time-series model and specifically the autoregression section for an explanation of the vectorization for `y` as well as an explanation of how to convert this example to a full AR(1) model. To ensure `y[1]` has a proper posterior in case it is missing, we have given it an explicit, albeit broad, prior.

Another potential application would be filling the columns of a data matrix of predictors for which some predictors are missing; matrix columns can be accessed as vectors and assigned the same way, as in

```
x[N_obs_2, 2] = x_obs_2;
x[N_mis_2, 2] = x_mis_2;
```

where the relevant variables are all hard coded with index `2` because Stan doesn't support ragged arrays. These could all be packed into a single array with more fiddly indexing that slices out vectors from longer vectors (see the ragged data structures section for a general discussion of coding ragged data structures in Stan).

3.4. Loading matrix for factor analysis

Rick Farouni, on the Stan users group, inquired as to how to build a Cholesky factor for a covariance matrix with a unit diagonal, as used in Bayesian factor analysis Aguilar and West (2000). This can be accomplished by declaring the below-diagonal elements as parameters, then filling the full matrix as a transformed parameter.

```
data {
  int<lower=2> K;
}
transformed data {
  int<lower=1> K_choose_2;
  K_choose_2 = (K * (K - 1)) / 2;
}
parameters {
  vector[K_choose_2] L_lower;
}
```

```
transformed parameters {
  cholesky_factor_cov[K] L;
  for (k in 1:K)
    L[k, k] = 1;
  {
    int i;
    for (m in 2:K) {
      for (n in 1:(m - 1)) {
        L[m, n] = L_lower[i];
        L[n, m] = 0;
        i += 1;
      }
    }
  }
}
```

It is most convenient to place a prior directly on `L_lower`. An alternative would be a prior for the full Cholesky factor `L`, because the transform from `L_lower` to `L` is just the identity and thus does not require a Jacobian adjustment (despite the warning from the parser, which is not smart enough to do the code analysis to infer that the transform is linear). It would not be at all convenient to place a prior on the full covariance matrix `L * L'`, because that would require a Jacobian adjustment; the exact adjustment is detailed in the reference manual.

3.5. Missing Multivariate Data

It's often the case that one or more components of a multivariate outcome are missing.[2]

As an example, we'll consider the bivariate distribution, which is easily marginalized. The coding here is brute force, representing both an array of vector observations `y` and a boolean array `y_observed` to indicate which values were observed (others can have dummy values in the input).

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

If both components are observed, we model them using the full multi-normal, otherwise we model the marginal distribution of the component that is observed.

[2]This is not the same as missing components of a multivariate predictor in a regression problem; in that case, you will need to represent the missing data as a parameter and impute missing values in order to feed them into the regression.

```
for (n in 1:N) {
  if (y_observed[n, 1] && y_observed[n, 2])
    y[n] ~ multi_normal(mu, Sigma);
  else if (y_observed[n, 1])
    y[n, 1] ~ normal(mu[1], sqrt(Sigma[1, 1]));
  else if (y_observed[n, 2])
    y[n, 2] ~ normal(mu[2], sqrt(Sigma[2, 2]));
}
```

It's a bit more work, but much more efficient to vectorize these sampling statements. In transformed data, build up three vectors of indices, for the three cases above:

```
transformed data {
  int ns12[observed_12(y_observed)];
  int ns1[observed_1(y_observed)];
  int ns2[observed_2(y_observed)];
}
```

You will need to write functions that pull out the count of observations in each of the three sampling situations. This must be done with functions because the result needs to go in top-level block variable size declaration. Then the rest of transformed data just fills in the values using three counters.

```
int n12 = 1;
int n1 = 1;
int n2 = 1;
for (n in 1:N) {
  if (y_observed[n, 1] && y_observed[n, 2]) {
    ns12[n12] = n;
    n12 += 1;
  } else if (y_observed[n, 1]) {
    ns1[n1] = n;
    n1 += 1;
  } else if (y_observed[n, 2]) {
    ns2[n2] = n;
    n2 += 1;
  }
}
```

Then, in the model block, everything is vectorizable using those indexes constructed once in transformed data:

```
y[ns12] ~ multi_normal(mu, Sigma);
y[ns1] ~ normal(mu[1], sqrt(Sigma[1, 1]));
y[ns2] ~ normal(mu[2], sqrt(Sigma[2, 2]));
```

The result will be much more efficient than using latent variables for the missing data, but it requires the multivariate distribution to be marginalized analytically. It'd be more efficient still to precompute the three arrays in the transformed data block, though the efficiency improvement will be relatively minor compared to vectorizing the probability functions.

This approach can easily be generalized with some index fiddling to the general multivariate case. The trick is to pull out entries in the covariance matrix for the missing components. It can also be used in situations such as multivariate differential equation solutions where only one component is observed, as in a phase-space experiment recording only time and position of a pendulum (and not recording momentum).

4. Truncated or Censored Data

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

4.1. Truncated Distributions

Truncation in Stan is restricted to univariate distributions for which the corresponding log cumulative distribution function (CDF) and log complementary cumulative distribution (CCDF) functions are available. See the reference manual section on truncated distributions for more information on truncated distributions, CDFs, and CCDFs.

4.2. Truncated Data

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

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

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

The model declares an upper bound `U` as data and constrains the data for `y` to respect

the constraint; this will be checked when the data are loaded into the model before sampling begins.

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

Constraints and Out-of-Bounds Returns

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

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

then if the value of `y[n]` is less than the value of `L` or greater than the value of `U`, the sampling statement produces a zero-probability estimate. For user-defined truncation, this zeroing outside of truncation bounds must be handled explicitly.

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

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

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

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

For pairs of real numbers, the function `fmax` is used rather than `max`.

Unknown Truncation Points

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

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

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

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

4.3. Censored Data

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

Estimating Censored Values

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

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

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

Integrating out Censored Values

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

$$\begin{aligned}\Pr[y > U] &= \int_U^{\infty} \mathsf{normal}\left(y \mid \mu, \sigma\right) \, \mathrm{d}y \\ &= 1 - \Phi\left(\frac{y - \mu}{\sigma}\right),\end{aligned}$$

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

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

where `normal_lccdf` is the log of complementary CDF (Stan provides `<distr>_lccdf` for each distribution implemented in Stan).

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

```
data {
  int<lower=0> N_obs;
  int<lower=0> N_cens;
  real y_obs[N_obs];
  real<lower=max(y_obs)> U;
}
parameters {
  real mu;
  real<lower=0> sigma;
}
model {
  y_obs ~ normal(mu, sigma);
  target += N_cens * normal_lccdf(U | mu, sigma);
}
```

For the observed values in `y_obs`, the normal sampling model is used without truncation. The log probability is directly incremented using the calculated log cumulative normal probability of the censored data items.

For the left-censored data the CDF (`normal_lcdf`) has to be used instead of complementary CDF. If the censoring point variable (`L`) is unknown, its declaration should be moved from the data to the parameters block.

```
data {
  int<lower=0> N_obs;
  int<lower=0> N_cens;
  real y_obs[N_obs];
}
parameters {
```

```
  real<upper=min(y_obs)> L;
  real mu;
  real<lower=0> sigma;
}
model {
  L ~ normal(mu, sigma);
  y_obs ~ normal(mu, sigma);
  target += N_cens * normal_lcdf(L | mu, sigma);
}
```

5. Finite Mixtures

Finite mixture models of an outcome assume that the outcome is drawn from one of several distributions, the identity of which is controlled by a categorical mixing distribution. Mixture models typically have multimodal densities with modes near the modes of the mixture components. Mixture models may be parameterized in several ways, as described in the following sections. Mixture models may be used directly for modeling data with multimodal distributions, or they may be used as priors for other parameters.

5.1. Relation to Clustering

Clustering models, as discussed in the clustering chapter, are just a particular class of mixture models that have been widely applied to clustering in the engineering and machine-learning literature. The normal mixture model discussed in this chapter reappears in multivariate form as the statistical basis for the K-means algorithm; the latent Dirichlet allocation model, usually applied to clustering problems, can be viewed as a mixed-membership multinomial mixture model.

5.2. Latent Discrete Parameterization

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

$$z_n \sim \mathsf{categorical}(\lambda).$$

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

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

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

5.3. Summing out the Responsibility Parameter

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

$$p_Y\left(y \mid \lambda, \mu, \sigma\right) = \sum_{k=1}^{K} \lambda_k \, \mathsf{normal}\left(y \mid \mu_k, \sigma_k\right).$$

Log Sum of Exponentials: Linear Sums on the Log Scale

The log sum of exponentials function is used to define mixtures on the log scale. It is defined for two inputs by

$$\texttt{log\_sum\_exp}(a, b) = \log\left(\exp(a) + \exp(b)\right).$$

If a and b are probabilities on the log scale, then $\exp(a) + \exp(b)$ is their sum on the linear scale, and the outer log converts the result back to the log scale; to summarize, log_sum_exp does linear addition on the log scale. The reason to use Stan's built-in `log_sum_exp` function is that it can prevent underflow and overflow in the exponentiation, by calculating the result as

$$\log\left(\exp(a) + \exp(b)\right) = c + \log\left(\exp(a - c) + \exp(b - c)\right),$$

where $c = \max(a, b)$. In this evaluation, one of the terms, $a - c$ or $b - c$, is zero and the other is negative, thus eliminating the possibility of overflow or underflow in the leading term while extracting the most arithmetic precision possible by pulling the $\max(a, b)$ out of the log-exp round trip.

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

```
parameters {
  real y;
}
model {
  target += log_sum_exp(log(0.3) + normal_lpdf(y | -1, 2),
                        log(0.7) + normal_lpdf(y | 3, 1));
}
```

The log probability term is derived by taking

$$
\begin{aligned}
&\log p\left(y \mid \lambda, \mu, \sigma\right) \\
&\quad = \log \left(0.3 \times \mathsf{normal}\left(y \mid -1, 2\right) + 0.7 \times \mathsf{normal}\left(y \mid 3, 1\right)\right) \\
&\quad = \log \left(\exp\left(\log\left(0.3 \times \mathsf{normal}\left(y \mid -1, 2\right)\right)\right) + \exp\left(\log\left(0.7 \times \mathsf{normal}\left(y \mid 3, 1\right)\right)\right)\right) \\
&\quad = \texttt{log\_sum\_exp}\big(\log(0.3) + \log \mathsf{normal}\left(y \mid -1, 2\right), \log(0.7) + \log \mathsf{normal}\left(y \mid 3, 1\right)\big).
\end{aligned}
$$

Dropping uniform mixture ratios

If a two-component mixture has a mixing ratio of 0.5, then the mixing ratios can be dropped, because

```
log_half = log(0.5);
for (n in 1:N)
  target += log_sum_exp(neg_log_half
                        + normal_lpdf(y[n] | mu[1], sigma[1]),
                        log_half
            + normal_lpdf(y[n] | mu[2], sigma[2]));
```

then the $\log 0.5$ term isn't contributing to the proportional density, and the above can be replaced with the more efficient version

```
for (n in 1:N)
  target += log_sum_exp(normal_lpdf(y[n] | mu[1], sigma[1]),
                        normal_lpdf(y[n] | mu[2], sigma[2]));
```

The same result holds if there are K components and the mixing simplex λ is symmetric, i.e.,

$$\lambda = \left(\frac{1}{K}, \ldots, \frac{1}{K}\right).$$

The result follows from the identity

$$\texttt{log\_sum\_exp}(c + a, c + b) = c + \texttt{log\_sum\_exp}(a, b)$$

and the fact that adding a constant c to the log density accumulator has no effect because the log density is only specified up to an additive constant in the first place. There is nothing specific to the normal distribution here; constants may always be dropped from the target.

Recovering posterior mixture proportions

The posterior $p(z_n \mid y_n, \mu, \sigma)$ over the mixture indicator $z_n \in 1:K$ is often of interest as $p(z_n = k \mid y, \mu, \sigma)$ is the posterior probability that that observation y_n was generated by mixture component k. The posterior can be computed via Bayes's rule,

$$\begin{aligned}\Pr\left(z_n = k \mid y_n, \mu, \sigma, \lambda\right) &\propto p\left(y_n \mid z_n = k, \mu, \sigma\right) p\left(z_n = k \mid \lambda\right) \\ &= \mathsf{normal}\left(y_n \mid \mu_k, \sigma_k\right) \cdot \lambda_k.\end{aligned}$$

The normalization can be done via summation, because $z_n \in 1{:}K$ only takes on finitely many values. In detail,

$$p\left(z_n = k \mid y_n, \mu, \sigma, \lambda\right) = \frac{p\left(y_n \mid z_n = k, \mu, \sigma\right) \cdot p\left(z_n = k \mid \lambda\right)}{\sum_{k'=1}^{K} p\left(y_n \mid z_n = k', \mu, \sigma\right) \cdot p\left(z_n = k' \mid \lambda\right)}.$$

On the log scale, the normalized probability is computed as

$$\begin{aligned}&\log \Pr\left(z_n = k \mid y_n, \mu, \sigma, \lambda\right) \\ &\quad = \log p\left(y_n \mid z_n = k, \mu, \sigma\right) + \log \Pr\left(z_n = k \mid \lambda\right) \\ &\qquad - \texttt{log\_sum\_exp}_{k'=1}^{K}\big(\log p\left(y_n \mid z_n = k', \mu, \sigma\right) + \log p\left(z_n = k' \mid \lambda\right)\big).\end{aligned}$$

This can be coded up directly in Stan; the change-point model in the change point section provides an example.

Estimating Parameters of a Mixture

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

```
data {
  int<lower=1> K;          // number of mixture components
  int<lower=1> N;          // number of data points
  real y[N];               // observations
}
parameters {
```

```
  simplex[K] theta;          // mixing proportions
  ordered[K] mu;             // locations of mixture components
  vector<lower=0>[K] sigma;  // scales of mixture components
}
model {
  vector[K] log_theta = log(theta);  // cache log calculation
  sigma ~ lognormal(0, 2);
  mu ~ normal(0, 10);
  for (n in 1:N) {
    vector[K] lps = log_theta;
    for (k in 1:K)
      lps[k] += normal_lpdf(y[n] | mu[k], sigma[k]);
    target += log_sum_exp(lps);
  }
}
```

The model involves `K` mixture components and `N` data points. The mixing proportion parameter `theta` is declared to be a unit K-simplex, whereas the component location parameter `mu` and scale parameter `sigma` are both defined to be `K`-vectors.

The location parameter `mu` is declared to be an ordered vector in order to identify the model. This will not affect inferences that do not depend on the ordering of the components as long as the prior for the components `mu[k]` is symmetric, as it is here (each component has an independent $\mathsf{normal}(0, 10)$ prior). It would even be possible to include a hierarchical prior for the components.

The values in the scale array `sigma` are constrained to be non-negative, and have a weakly informative prior given in the model chosen to avoid zero values and thus collapsing components.

The model declares a local array variable `lps` to be size `K` and uses it to accumulate the log contributions from the mixture components. The main action is in the loop over data points `n`. For each such point, the log of $\theta_k \times \mathsf{normal}\,(y_n \mid \mu_k, \sigma_k)$ is calculated and added to the array `lpps`. Then the log probability is incremented with the log sum of exponentials of those values.

5.4. Vectorizing Mixtures

There is (currently) no way to vectorize mixture models at the observation level in Stan. This section is to warn users away from attempting to vectorize naively, as it results in a different model. A proper mixture at the observation level is defined

as follows, where we assume that `lambda`, `y[n]`, `mu[1]`, `mu[2]`, and `sigma[1]`, `sigma[2]` are all scalars and `lambda` is between 0 and 1.

```
for (n in 1:N) {
  target += log_sum_exp(log(lambda)
                          + normal_lpdf(y[n] | mu[1], sigma[1]),
                        log1m(lambda)
                          + normal_lpdf(y[n] | mu[2], sigma[2]));
```

or equivalently

```
for (n in 1:N)
  target += log_mix(lambda,
                    normal_lpdf(y[n] | mu[1], sigma[1]),
                    normal_lpdf(y[n] | mu[2], sigma[2]));
```

This definition assumes that each observation y_n may have arisen from either of the mixture components. The density is

$$p\left(y \mid \lambda, \mu, \sigma\right) = \prod_{n=1}^{N} \left(\lambda \times \mathsf{normal}\left(y_n \mid \mu_1, \sigma_1\right) + (1 - \lambda) \times \mathsf{normal}\left(y_n \mid \mu_2, \sigma_2\right)\right).$$

Contrast the previous model with the following (erroneous) attempt to vectorize the model.

```
target += log_sum_exp(log(lambda)
                        + normal_lpdf(y | mu[1], sigma[1]),
                      log1m(lambda)
                        + normal_lpdf(y | mu[2], sigma[2]));
```

or equivalently,

```
target += log_mix(lambda,
                  normal_lpdf(y | mu[1], sigma[1]),
                  normal_lpdf(y | mu[2], sigma[2]));
```

This second definition implies that the entire sequence $y_1, \ldots, y_n$ of observations comes form one component or the other, defining a different density,

$$p\left(y \mid \lambda, \mu, \sigma\right) = \lambda \times \prod_{n=1}^{N} \mathsf{normal}\left(y_n \mid \mu_1, \sigma_1\right) + (1 - \lambda) \times \prod_{n=1}^{N} \mathsf{normal}\left(y_n \mid \mu_2, \sigma_2\right).$$

5.5. Inferences Supported by Mixtures

In many mixture models, the mixture components are underlyingly exchangeable in the model and thus not identifiable. This arises if the parameters of the mixture components have exchangeable priors and the mixture ratio gets a uniform prior so that the parameters of the mixture components are also exchangeable in the likelihood.

We have finessed this basic problem by ordering the parameters. This will allow us in some cases to pick out mixture components either ahead of time or after fitting (e.g., male vs. female, or Democrat vs. Republican).

In other cases, we do not care about the actual identities of the mixture components and want to consider inferences that are independent of indexes. For example, we might only be interested in posterior predictions for new observations.

Mixtures with Unidentifiable Components

As an example, consider the normal mixture from the previous section, which provides an exchangeable prior on the pairs of parameters (μ_1, σ_1) and (μ_2, σ_2),

$$\mu_1, \mu_2 \sim \mathsf{normal}(0, 10)$$
$$\sigma_1, \sigma_2 \sim \mathsf{halfnormal}(0, 10)$$

The prior on the mixture ratio is uniform,

$$\lambda \sim \mathsf{uniform}(0, 1),$$

so that with the likelihood

$$p\left(y_n \mid \mu, \sigma\right) = \lambda \times \mathsf{normal}\left(y_n \mid \mu_1, \sigma_1\right) + (1 - \lambda) \times \mathsf{normal}\left(y_n \mid \mu_2, \sigma_2\right),$$

the joint distribution $p(y, \mu, \sigma, \lambda)$ is exchangeable in the parameters (μ_1, σ_1) and (μ_2, σ_2) with λ flipping to $1 - \lambda$.[1]

[1]Imposing a constraint such as $\theta < 0.5$ will resolve the symmetry, but fundamentally changes the model and its posterior inferences.

Inference under Label Switching

In cases where the mixture components are not identifiable, it can be difficult to diagnose convergence of sampling or optimization algorithms because the labels will switch, or be permuted, in different MCMC chains or different optimization runs. Luckily, posterior inferences which do not refer to specific component labels are invariant under label switching and may be used directly. This subsection considers a pair of examples.

Predictive likelihood

Predictive likelihood for a new observation $\tilde{y}$ given the complete parameter vector θ will be

$$p(\tilde{y} \mid y) = \int_{\theta} p(\tilde{y} \mid \theta)\, p(\theta \mid y)\, \mathsf{d}\theta.$$

The normal mixture example from the previous section, with $\theta = (\mu, \sigma, \lambda)$, shows that the likelihood returns the same density under label switching and thus the predictive inference is sound. In Stan, that predictive inference can be done either by computing $p(\tilde{y} \mid y)$, which is more efficient statistically in terms of effective sample size, or simulating draws of $\tilde{y}$, which is easier to plug into other inferences. Both approaches can be coded directly in the generated quantities block of the program. Here's an example of the direct (non-sampling) approach.

```
data {
  int<lower = 0> N_tilde;
  vector[N_tilde] y_tilde;
  ...
generated quantities {
  vector[N_tilde] log_p_y_tilde;
  for (n in 1:N_tilde)
    log_p_y_tilde[n]
      = log_mix(lambda,
                normal_lpdf(y_tilde[n] | mu[1], sigma[1])
                normal_lpdf(y_tilde[n] | mu[2], sigma[2]));
}
```

It is a bit of a bother afterwards, because the logarithm function isn't linear and hence doesn't distribute through averages (Jensen's inequality shows which way the inequality goes). The right thing to do is to apply `log_sum_exp` of the posterior draws

of `log_p_y_tilde`. The average log predictive density is then given by subtracting `log(N_new)`.

Clustering and similarity

Often a mixture model will be applied to a clustering problem and there might be two data items y_i and y_j for which there is a question of whether they arose from the same mixture component. If we take z_i and z_j to be the component responsibility discrete variables, then the quantity of interest is $z_i = z_j$, which can be summarized as an event probability

$$\Pr[z_i = z_j \mid y] = \int_\theta \frac{\sum_{k=0}^{1} p(z_i = k, z_j = k, y_i, y_j \mid \theta)}{\sum_{k=0}^{1} \sum_{m=0}^{1} p(z_i = k, z_j = m, y_i, y_j \mid \theta)} \, p(\theta \mid y) \, \mathsf{d}\theta.$$

As with other event probabilities, this can be calculated in the generated quantities block either by sampling z_i and z_j and using the indicator function on their equality, or by computing the term inside the integral as a generated quantity. As with predictive likelihood, working in expectation is more statistically efficient than sampling.

5.6. Zero-Inflated and Hurdle Models

Zero-inflated and hurdle models both provide mixtures of a Poisson and Bernoulli probability mass function to allow more flexibility in modeling the probability of a zero outcome. Zero-inflated models, as defined by Lambert (1992), add additional probability mass to the outcome of zero. Hurdle models, on the other hand, are formulated as pure mixtures of zero and non-zero outcomes.

Zero inflation and hurdle models can be formulated for discrete distributions other than the Poisson. Zero inflation does not work for continuous distributions in Stan because of issues with derivatives; in particular, there is no way to add a point mass to a continuous distribution, such as zero-inflating a normal as a regression coefficient prior.

Zero Inflation

Consider the following example for zero-inflated Poisson distributions. It uses a parameter `theta` here there is a probability θ of drawing a zero, and a probability $1-\theta$ of drawing from $\mathsf{Poisson}(\lambda)$ (now θ is being used for mixing proportions because

λ is the traditional notation for a Poisson mean parameter). The probability function is thus

$$p(y_n \mid \theta, \lambda) = \begin{cases} \theta + (1 - \theta) \times \mathsf{Poisson}(0 \mid \lambda) & \text{if } y_n = 0, \text{ and} \\ (1 - \theta) \times \mathsf{Poisson}(y_n \mid \lambda) & \text{if } y_n > 0. \end{cases}$$

The log probability function can be implemented directly in Stan as follows.

```
data {
  int<lower=0> N;
  int<lower=0> y[N];
}
parameters {
  real<lower=0, upper=1> theta;
  real<lower=0> lambda;
}
model {
  for (n in 1:N) {
    if (y[n] == 0)
      target += log_sum_exp(bernoulli_lpmf(1 | theta),
                            bernoulli_lpmf(0 | theta)
                              + poisson_lpmf(y[n] | lambda));
    else
      target += bernoulli_lpmf(0 | theta)
                  + poisson_lpmf(y[n] | lambda);
  }
}
```

The `log_sum_exp(lp1,lp2)` function adds the log probabilities on the linear scale; it is defined to be equal to `log(exp(lp1) + exp(lp2))`, but is more arithmetically stable and faster.

Optimizing the zero-inflated Poisson model

The code given above to compute the zero-inflated Poisson redundantly calculates all of the Bernoulli terms and also `poisson_lpmf(0 \mid lambda)` every time the first condition body executes. The use of the redundant terms is conditioned on `y`, which is known when the data are read in. This allows the transformed data block to be used to compute some more convenient terms for expressing the log density each iteration.

The number of zero cases is computed and handled separately. Then the nonzero cases are collected into their own array for vectorization. The number of zeros is required to declare `y_nonzero`, so it must be computed in a function.

```
functions {
  int num_zeros(int[] y) {
    int sum = 0;
    for (n in 1:size(y))
      sum += (y[n] == 0);
    return sum;
  }
}
...
transformed data {
  int<lower = 0> N_zero = num_zeros(y);
  int<lower = 1> y_nonzero[N - N_zero];
  int N_nonzero = 0;
  for (n in 1:N) {
    if (y[n] == 0) continue;
    N_nonzero += 1;
    y_nonzero[N_nonzero] = y[n];
  }
}
...
model {
  ...
   target
     += N_zero
          * log_sum_exp(bernoulli_lpmf(1 | theta),
                        bernoulli_lpmf(0 | theta)
                          + poisson_lpmf(0 | lambda));
   target += N_nonzero * bernoulli_lpmf(0 | theta);
   target += poisson_lpmf(y_nonzero | lambda);
...
```

The boundary conditions of all zeros and no zero outcomes is handled appropriately; in the vectorized case, if `y_nonzero` is empty, `N_nonzero` will be zero, and the last two target increment terms will add zeros.

Hurdle Models

The hurdle model is similar to the zero-inflated model, but more flexible in that the zero outcomes can be deflated as well as inflated. The probability mass function for the hurdle likelihood is defined by

$$p(y \mid \theta, \lambda) = \begin{cases} \theta & \text{if } y = 0, \text{ and} \\ (1 - \theta) \dfrac{\mathsf{Poisson}(y \mid \lambda)}{1 - \mathsf{PoissonCDF}(0 \mid \lambda)} & \text{if } y > 0, \end{cases}$$

where PoissonCDF is the cumulative distribution function for the Poisson distribution. The hurdle model is even more straightforward to program in Stan, as it does not require an explicit mixture.

```
if (y[n] == 0)
  1 ~ bernoulli(theta);
else {
  0 ~ bernoulli(theta);
  y[n] ~ poisson(lambda) T[1, ];
}
```

The Bernoulli statements are just shorthand for adding $\log \theta$ and $\log(1 - \theta)$ to the log density. The `T[1,]` after the Poisson indicates that it is truncated below at 1; see the truncation section for more about truncation and the Poisson regression section for the specifics of the Poisson CDF. The net effect is equivalent to the direct definition of the log likelihood.

```
if (y[n] == 0)
  target += log(theta);
else
  target += log1m(theta) + poisson_lpmf(y[n] | lambda)
            - poisson_lccdf(0 | lambda));
```

Julian King pointed out that because

$$\begin{aligned} \log\left(1 - \mathsf{PoissonCDF}(0 \mid \lambda)\right) &= \log\left(1 - \mathsf{Poisson}(0 \mid \lambda)\right) \\ &= \log(1 - \exp(-\lambda)) \end{aligned}$$

the CCDF in the else clause can be replaced with a simpler expression.

```
target += log1m(theta) + poisson_lpmf(y[n] | lambda)
          - log1m_exp(-lambda));
```

The resulting code is about 15% faster than the code with the CCDF.

This is an example where collecting counts ahead of time can also greatly speed up the execution speed without changing the density. For data size $N = 200$ and parameters $\theta = 0.3$ and $\lambda = 8$, the speedup is a factor of 10; it will be lower for smaller N and greater for larger N; it will also be greater for larger θ.

To achieve this speedup, it helps to have a function to count the number of non-zero entries in an array of integers,

```
functions {
  int num_zero(int[] y) {
    int nz = 0;
    for (n in 1:size(y))
      if (y[n] == 0)
        nz += 1;
    return nz;
  }
}
```

Then a transformed data block can be used to store the sufficient statistics,

```
transformed data {
  int<lower=0, upper=N> N0 = num_zero(y);
  int<lower=0, upper=N> Ngt0 = N - N0;
  int<lower=1> y_nz[N - num_zero(y)];
  {
    int pos = 1;
    for (n in 1:N) {
      if (y[n] != 0) {
        y_nz[pos] = y[n];
        pos += 1;
      }
    }
  }
}
```

The model block can then be reduced to three statements.

```
model {
  N0 ~ binomial(N, theta);
  y_nz ~ poisson(lambda);
  target += -Ngt0 * log1m_exp(-lambda);
```

```
}
```

The first statement accounts for the Bernoulli contribution to both the zero and non-zero counts. The second line is the Poisson contribution from the non-zero counts, which is now vectorized. Finally, the normalization for the truncation is a single line, so that the expression for the log CCDF at 0 isn't repeated. Also note that the negation is applied to the constant `Ngt0`; whenever possible, leave subexpressions constant because then gradients need not be propagated until a non-constant term is encountered.

5.7. Priors and Effective Data Size in Mixture Models

Suppose we have a two-component mixture model with mixing rate $\lambda \in (0, 1)$. Because the likelihood for the mixture components is proportionally weighted by the mixture weights, the effective data size used to estimate each of the mixture components will also be weighted as a fraction of the overall data size. Thus although there are N observations, the mixture components will be estimated with effective data sizes of $\theta\, N$ and $(1 - \theta)\, N$ for the two components for some $\theta \in (0, 1)$. The effective weighting size is determined by posterior responsibility, not simply by the mixing rate λ.

Comparison to Model Averaging

In contrast to mixture models, which create mixtures at the observation level, model averaging creates mixtures over the posteriors of models separately fit with the entire data set. In this situation, the priors work as expected when fitting the models independently, with the posteriors being based on the complete observed data y.

If different models are expected to account for different observations, we recommend building mixture models directly. If the models being mixed are similar, often a single expanded model will capture the features of both and may be used on its own for inferential purposes (estimation, decision making, prediction, etc.). For example, rather than fitting an intercept-only regression and a slope-only regression and averaging their predictions, even as a mixture model, we would recommend building a single regression with both a slope and an intercept. Model complexity, such as having more predictors than data points, can be tamed using appropriately regularizing priors. If computation becomes a bottleneck, the only recourse can be model averaging, which can be calculated after fitting each model independently (see Hoeting et al. (1999) and Gelman et al. (2013) for theoretical and computational details).

6. Measurement Error and Meta-Analysis

Most quantities used in statistical models arise from measurements. Most of these measurements are taken with some error. When the measurement error is small relative to the quantity being measured, its effect on a model is usually small. When measurement error is large relative to the quantity being measured, or when precise relations can be estimated being measured quantities, it is useful to introduce an explicit model of measurement error. One kind of measurement error is rounding.

Meta-analysis plays out statistically much like measurement error models, where the inferences drawn from multiple data sets are combined to do inference over all of them. Inferences for each data set are treated as providing a kind of measurement error with respect to true parameter values.

6.1. Bayesian Measurement Error Model

A Bayesian approach to measurement error can be formulated directly by treating the true quantities being measured as missing data (Clayton 1992; Richardson and Gilks 1993). This requires a model of how the measurements are derived from the true values.

Regression with Measurement Error

Before considering regression with measurement error, first consider a linear regression model where the observed data for N cases includes a predictor x_n and outcome y_n. In Stan, a linear regression for y based on x with a slope and intercept is modeled as follows.

```
data {
  int<lower=0> N;       // number of cases
  vector[N] x;          // predictor (covariate)
  vector[N] y;          // outcome (variate)
}
parameters {
  real alpha;           // intercept
  real beta;            // slope
  real<lower=0> sigma;  // outcome noise
}
```

```
model {
  y ~ normal(alpha + beta * x, sigma);
  alpha ~ normal(0, 10);
  beta ~ normal(0, 10);
  sigma ~ cauchy(0, 5);
}
```

Now suppose that the true values of the predictors x_n are not known, but for each n, a measurement x_n^{meas} of x_n is available. If the error in measurement can be modeled, the measured value x_n^{meas} can be modeled in terms of the true value x_n plus measurement noise. The true value x_n is treated as missing data and estimated along with other quantities in the model. A simple approach is to assume the measurement error is normal with known deviation τ. This leads to the following regression model with constant measurement error.

```
data {
  ...
  real x_meas[N];     // measurement of x
  real<lower=0> tau;  // measurement noise
}
parameters {
  real x[N];          // unknown true value
  real mu_x;          // prior location
  real sigma_x;       // prior scale
  ...
}
model {
  x ~ normal(mu_x, sigma_x);  // prior
  x_meas ~ normal(x, tau);    // measurement model
  y ~ normal(alpha + beta * x, sigma);
  ...
}
```

The regression coefficients `alpha` and `beta` and regression noise scale `sigma` are the same as before, but now `x` is declared as a parameter rather than as data. The data are now `x_meas`, which is a measurement of the true `x` value with noise scale `tau`. The model then specifies that the measurement error for `x_meas[n]` given true value `x[n]` is normal with deviation `tau`. Furthermore, the true values `x` are given a hierarchical prior here.

In cases where the measurement errors are not normal, richer measurement error

models may be specified. The prior on the true values may also be enriched. For instance, Clayton (1992) introduces an exposure model for the unknown (but noisily measured) risk factors x in terms of known (without measurement error) risk factors c. A simple model would regress x_n on the covariates c_n with noise term v,

$$x_n \sim \mathsf{normal}(\gamma^\top c, v).$$

This can be coded in Stan just like any other regression. And, of course, other exposure models can be provided.

Rounding

A common form of measurement error arises from rounding measurements. Rounding may be done in many ways, such as rounding weights to the nearest milligram, or to the nearest pound; rounding may even be done by rounding down to the nearest integer.

Exercise 3.5(b) from Gelman et al. (2013) provides an example.

> 3.5. Suppose we weigh an object five times and measure weights, rounded to the nearest pound, of 10, 10, 12, 11, 9. Assume the unrounded measurements are normally distributed with a noninformative prior distribution on μ and σ^2.
>
> (b) Give the correct posterior distribution for (μ, σ^2), treating the measurements as rounded.

Letting z_n be the unrounded measurement for y_n, the problem as stated assumes the likelihood

$$z_n \sim \mathsf{normal}(\mu, \sigma).$$

The rounding process entails that $z_n \in (y_n - 0.5, y_n + 0.5)$. The probability mass function for the discrete observation y is then given by marginalizing out the unrounded measurement, producing the likelihood

$$\begin{aligned} p(y_n \mid \mu, \sigma) &= \int_{y_n - 0.5}^{y_n + 0.5} \mathsf{normal}(z_n \mid \mu, \sigma) \, \mathrm{d}z_n \\ &= \Phi\left(\frac{y_n + 0.5 - \mu}{\sigma}\right) - \Phi\left(\frac{y_n - 0.5 - \mu}{\sigma}\right). \end{aligned}$$

Gelman's answer for this problem took the noninformative prior to be uniform in the variance σ^2 on the log scale, which yields (due to the Jacobian adjustment), the

prior density

$$p(\mu, \sigma^2) \propto \frac{1}{\sigma^2}.$$

The posterior after observing $y = (10, 10, 12, 11, 9)$ can be calculated by Bayes's rule as

$$\begin{aligned} p(\mu, \sigma^2 \mid y) &\propto p(\mu, \sigma^2)\, p(y \mid \mu, \sigma^2) \\ &\propto \frac{1}{\sigma^2} \prod_{n=1}^{5} \left(\Phi\left(\frac{y_n + 0.5 - \mu}{\sigma} \right) - \Phi\left(\frac{y_n - 0.5 - \mu}{\sigma} \right) \right). \end{aligned}$$

The Stan code simply follows the mathematical definition, providing an example of the direct definition of a probability function up to a proportion.

```
data {
  int<lower=0> N;
  vector[N] y;
}
parameters {
  real mu;
  real<lower=0> sigma_sq;
}
transformed parameters {
  real<lower=0> sigma;
  sigma = sqrt(sigma_sq);
}
model {
  target += -2 * log(sigma);
  for (n in 1:N)
    target += log(Phi((y[n] + 0.5 - mu) / sigma)
                  - Phi((y[n] - 0.5 - mu) / sigma));
}
```

Alternatively, the model may be defined with latent parameters for the unrounded measurements z_n. The Stan code in this case uses the likelihood for z_n directly while respecting the constraint $z_n \in (y_n - 0.5, y_n + 0.5)$. Because Stan does not allow varying upper- and lower-bound constraints on the elements of a vector (or array), the parameters are declared to be the rounding error $y - z$, and then z is defined as a transformed parameter.

```
data {
  int<lower=0> N;
  vector[N] y;
}
parameters {
  real mu;
  real<lower=0> sigma_sq;
  vector<lower=-0.5, upper=0.5>[N] y_err;
}
transformed parameters {
  real<lower=0> sigma;
  vector[N] z;
  sigma = sqrt(sigma_sq);
  z = y + y_err;
}
model {
  target += -2 * log(sigma);
  z ~ normal(mu, sigma);
}
```

This explicit model for the unrounded measurements z produces the same posterior for μ and σ as the previous model that marginalizes z out. Both approaches mix well, but the latent parameter version is about twice as efficient in terms of effective samples per iteration, as well as providing a posterior for the unrounded parameters.

6.2. Meta-Analysis

Meta-analysis aims to pool the data from several studies, such as the application of a tutoring program in several schools or treatment using a drug in several clinical trials.

The Bayesian framework is particularly convenient for meta-analysis, because each previous study can be treated as providing a noisy measurement of some underlying quantity of interest. The model then follows directly from two components, a prior on the underlying quantities of interest and a measurement-error style model for each of the studies being analyzed.

Treatment Effects in Controlled Studies

Suppose the data in question arise from a total of M studies providing paired binomial data for a treatment and control group. For instance, the data might be

post-surgical pain reduction under a treatment of ibuprofen Warn, Thompson, and Spiegelhalter (2002) or mortality after myocardial infarction under a treatment of beta blockers (Gelman et al. 2013, Section 5.6).

Data

The clinical data consists of J trials, each with n^t treatment cases, n^c control cases, r^t successful outcomes among those treated and r^c successful outcomes among those in the control group. This data can be declared in Stan as follows.[1]

```
data {
  int<lower=0> J;
  int<lower=0> n_t[J];  // num cases, treatment
  int<lower=0> r_t[J];  // num successes, treatment
  int<lower=0> n_c[J];  // num cases, control
  int<lower=0> r_c[J];  // num successes, control
}
```

Converting to Log Odds and Standard Error

Although the clinical trial data are binomial in its raw format, it may be transformed to an unbounded scale by considering the log odds ratio

$$\begin{aligned} y_j &= \log \left(\frac{r_j^t / (n_j^t - r_j^t)}{r_j^c / (n_j^c - r_j^c)} \right) \\ &= \log \left(\frac{r_j^t}{n_j^t - r_j^t} \right) - \log \left(\frac{r_j^c}{n_j^c - r_j^c} \right) \end{aligned}$$

and corresponding standard errors

$$\sigma_j = \sqrt{\frac{1}{r_i^T} + \frac{1}{n_i^T - r_i^T} + \frac{1}{r_i^C} + \frac{1}{n_i^C - r_i^C}}.$$

[1] Stan's integer constraints are not powerful enough to express the constraint that `r_t[j]` $\leq$ `n_t[j]`, but this constraint could be checked in the transformed data block.

The log odds and standard errors can be defined in a transformed parameter block, though care must be taken not to use integer division.[2]

```
transformed data {
  real y[J];
  real<lower=0> sigma[J];
  for (j in 1:J)
    y[j] = log(r_t[j]) - log(n_t[j] - r_t[j])
           - (log(r_c[j]) - log(n_c[j] - r_c[j]);
  for (j in 1:J)
    sigma[j] = sqrt(1 / r_t[j] + 1 / (n_t[j] - r_t[j])
                    + 1 / r_c[j] + 1 / (n_c[j] - r_c[j]));
}
```

This definition will be problematic if any of the success counts is zero or equal to the number of trials. If that arises, a direct binomial model will be required or other transforms must be used than the unregularized sample log odds.

Non-Hierarchical Model

With the transformed data in hand, two standard forms of meta-analysis can be applied. The first is a so-called "fixed effects" model, which assumes a single parameter for the global odds ratio. This model is coded in Stan as follows.

```
parameters {
  real theta;  // global treatment effect, log odds
}
model {
  y ~ normal(theta, sigma);
}
```

The sampling statement for `y` is vectorized; it has the same effect as the following.

```
  for (j in 1:J)
    y[j] ~ normal(theta, sigma[j]);
```

It is common to include a prior for `theta` in this model, but it is not strictly necessary for the model to be proper because `y` is fixed and $\mathsf{normal}(y \mid \mu, \sigma) = \mathsf{normal}(\mu \mid y, \sigma)$.

[2] When dividing two integers, the result type is an integer and rounding will ensue if the result is not exact. See the discussion of primitive arithmetic types in the reference manual for more information.

Hierarchical Model

To model so-called "random effects," where the treatment effect may vary by clinical trial, a hierarchical model can be used. The parameters include per-trial treatment effects and the hierarchical prior parameters, which will be estimated along with other unknown quantities.

```
parameters {
  real theta[J];      // per-trial treatment effect
  real mu;            // mean treatment effect
  real<lower=0> tau;  // deviation of treatment effects
}
model {
  y ~ normal(theta, sigma);
  theta ~ normal(mu, tau);
  mu ~ normal(0, 10);
  tau ~ cauchy(0, 5);
}
```

Although the vectorized sampling statement for `y` appears unchanged, the parameter `theta` is now a vector. The sampling statement for `theta` is also vectorized, with the hyperparameters `mu` and `tau` themselves being given wide priors compared to the scale of the data.

Rubin (1981) provided a hierarchical Bayesian meta-analysis of the treatment effect of Scholastic Aptitude Test (SAT) coaching in eight schools based on the sample treatment effect and standard error in each school.

Extensions and Alternatives

Smith, Spiegelhalter, and Thomas (1995) and (Gelman et al. 2013, Section 19.4) provides meta-analyses based directly on binomial data. Warn, Thompson, and Spiegelhalter (2002) consider the modeling implications of using alternatives to the log-odds ratio in transforming the binomial data.

If trial-specific predictors are available, these can be included directly in a regression model for the per-trial treatment effects θ_j.

7. Latent Discrete Parameters

Stan does not support sampling discrete parameters. So it is not possible to directly translate BUGS or JAGS models with discrete parameters (i.e., discrete stochastic nodes). Nevertheless, it is possible to code many models that involve bounded discrete parameters by marginalizing out the discrete parameters.[1]

This chapter shows how to code several widely-used models involving latent discrete parameters. The next chapter, the clustering chapter, on clustering models, considers further models involving latent discrete parameters.

7.1. The Benefits of Marginalization

Although it requires some algebra on the joint probability function, a pleasant byproduct of the required calculations is the posterior expectation of the marginalized variable, which is often the quantity of interest for a model. This allows far greater exploration of the tails of the distribution as well as more efficient sampling on an iteration-by-iteration basis because the expectation at all possible values is being used rather than itself being estimated through sampling a discrete parameter.

Standard optimization algorithms, including expectation maximization (EM), are often provided in applied statistics papers to describe maximum likelihood estimation algorithms. Such derivations provide exactly the marginalization needed for coding the model in Stan.

7.2. Change Point Models

The first example is a model of coal mining disasters in the U.K. for the years 1851–1962.[2]

Model with Latent Discrete Parameter

(Fonnesbeck et al. 2013 Section 3.1) provides a Poisson model of disaster D_t in year t with two rate parameters, an early rate (e) and late rate (l), that change at a given point in time s. The full model expressed using a latent discrete parameter s is

[1] The computations are similar to those involved in expectation maximization (EM) algorithms Dempster, Laird, and Rubin (1977).

[2] The source of the data is Jarrett (1979), which itself is a note correcting an earlier data collection.

$$
\begin{aligned}
e &\sim \text{exponential}(r_e) \\
l &\sim \text{exponential}(r_l) \\
s &\sim \text{uniform}(1, T) \\
D_t &\sim \text{Poisson}(t < s \ ? \ e \ : \ l)
\end{aligned}
$$

The last line uses the conditional operator (also known as the ternary operator), which is borrowed from C and related languages. The conditional operator has the same behavior as its counterpart in C++.[3]

It uses a compact notation involving separating its three arguments by a question mark (`?`) and a colon (`:`). The conditional operator is defined by

$$
c \ ? \ x_1 \ : \ x_2 = \begin{cases} x_1 & \text{if } c \text{ is true (i.e., non-zero), and} \\ x_2 & \text{if } c \text{ is false (i.e., zero).} \end{cases}
$$

Marginalizing out the Discrete Parameter

To code this model in Stan, the discrete parameter s must be marginalized out to produce a model defining the log of the probability function $p(e, l, D_t)$. The full joint probability factors as

$$
\begin{aligned}
p(e, l, s, D) &= p(e)\, p(l)\, p(s)\, p(D \mid s, e, l) \\
&= \text{exponential}(e \mid r_e)\ \text{exponential}(l \mid r_l)\ \text{uniform}(s \mid 1, T) \\
&\quad \prod_{t=1}^{T} \text{Poisson}(D_t \mid t < s \ ? \ e \ : \ l),
\end{aligned}
$$

To marginalize, an alternative factorization into prior and likelihood is used,

$$
p(e, l, D) = p(e, l)\, p(D \mid e, l),
$$

where the likelihood is defined by marginalizing s as

[3]The R counterpart, `ifelse`, is slightly different in that it is typically used in a vectorized situation. The conditional operator is not (yet) vectorized in Stan.

$$
\begin{aligned}
p(D \mid e, l) &= \sum_{s=1}^{T} p(s, D \mid e, l) \\
&= \sum_{s=1}^{T} p(s)\, p(D \mid s, e, l) \\
&= \sum_{s=1}^{T} \mathsf{uniform}(s \mid 1, T) \prod_{t=1}^{T} \mathsf{Poisson}(D_t \mid t < s \ ? \ e \ : \ l)
\end{aligned}
$$

Stan operates on the log scale and thus requires the log likelihood,

$$
\begin{aligned}
\log p(D \mid e, l) = \texttt{log\_sum\_exp}_{s=1}^{T} \Bigg(& \log \mathsf{uniform}(s \mid 1, T) \\
& + \sum_{t=1}^{T} \log \mathsf{Poisson}(D_t \mid t < s \ ? \ e \ : \ l) \Bigg),
\end{aligned}
$$

where the log sum of exponents function is defined by

$$
\texttt{log\_sum\_exp}_{n=1}^{N} \alpha_n = \log \sum_{n=1}^{N} \exp(\alpha_n).
$$

The log sum of exponents function allows the model to be coded directly in Stan using the built-in function `log_sum_exp`, which provides both arithmetic stability and efficiency for mixture model calculations.

Coding the Model in Stan

The Stan program for the change point model is shown in the figure below. The transformed parameter `lp[s]` stores the quantity $\log p(s, D \mid e, l)$.

```
data {
  real<lower=0> r_e;
  real<lower=0> r_l;

  int<lower=1> T;
  int<lower=0> D[T];
```

```
}
transformed data {
  real log_unif;
  log_unif = -log(T);
}
parameters {
  real<lower=0> e;
  real<lower=0> l;
}
transformed parameters {
  vector[T] lp;
  lp = rep_vector(log_unif, T);
  for (s in 1:T)
    for (t in 1:T)
      lp[s] = lp[s] + poisson_lpmf(D[t] | t < s ? e : l);
}
model {
  e ~ exponential(r_e);
  l ~ exponential(r_l);
  target += log_sum_exp(lp);
}
```

A change point model in which disaster rates `D[t]` have one rate, `e`, before the change point and a different rate, `l`, after the change point. The change point itself, `s`, is marginalized out as described in the text.

Although the change-point model is coded directly, the doubly nested loop used for `s` and `t` is quadratic in `T`. Luke Wiklendt pointed out that a linear alternative can be achieved by the use of dynamic programming similar to the forward-backward algorithm for Hidden Markov models; he submitted a slight variant of the following code to replace the transformed parameters block of the above Stan program.

```
transformed parameters {
    vector[T] lp;
    {
      vector[T + 1] lp_e;
      vector[T + 1] lp_l;
      lp_e[1] = 0;
      lp_l[1] = 0;
      for (t in 1:T) {
        lp_e[t + 1] = lp_e[t] + poisson_lpmf(D[t] | e);
```

```
        lp_l[t + 1] = lp_l[t] + poisson_lpmf(D[t] | l);
      }
      lp = rep_vector(log_unif + lp_l[T + 1], T)
           + head(lp_e, T) - head(lp_l, T);
    }
  }
```

As should be obvious from looking at it, it has linear complexity in `T` rather than quadratic. The result for the mining-disaster data is about 20 times faster; the improvement will be greater for larger `T`.

The key to understanding Wiklendt's dynamic programming version is to see that `head(lp_e)` holds the forward values, whereas `lp_l[T + 1] - head(lp_l, T)` holds the backward values; the clever use of subtraction allows `lp_l` to be accumulated naturally in the forward direction.

Fitting the Model with MCMC

This model is easy to fit using MCMC with NUTS in its default configuration. Convergence is fast and sampling produces roughly one effective sample every two iterations. Because it is a relatively small model (the inner double loop over time is roughly 20,000 steps), it is fast.

The value of `lp` for each iteration for each change point is available because it is declared as a transformed parameter. If the value of `lp` were not of interest, it could be coded as a local variable in the model block and thus avoid the I/O overhead of saving values every iteration.

Posterior Distribution of the Discrete Change Point

The value of `lp[s]` in a given iteration is given by $\log p(s, D \mid e, l)$ for the values of the early and late rates, e and l, in the iteration. In each iteration after convergence, the early and late disaster rates, e and l, are drawn from the posterior $p(e, l \mid D)$ by MCMC sampling and the associated `lp` calculated. The value of `lp` may be normalized to calculate $p(s \mid e, l, D)$ in each iteration, based on on the current values of e and l. Averaging over iterations provides an unnormalized probability estimate of the change point being s (see below for the normalizing constant),

$$
\begin{aligned}
p(s \mid D) &\propto q(s \mid D) \\
&= \frac{1}{M} \sum_{m=1}^{M} \exp(\texttt{lp}[m, s]).
\end{aligned}
$$

where $\texttt{lp}[m, s]$ represents the value of `lp` in posterior draw m for change point s. By averaging over draws, e and l are themselves marginalized out, and the result has no dependence on a given iteration's value for e and l. A final normalization then produces the quantity of interest, the posterior probability of the change point being s conditioned on the data D,

$$
p(s \mid D) = \frac{q(s \mid D)}{\sum_{s'=1}^{T} q(s' \mid D)}.
$$

A plot of the values of $\log p(s \mid D)$ computed using Stan 2.4's default MCMC implementation is shown in the posterior plot.

Log probability of change point being in year, calculated analytically.

The frequency of change points generated by sampling the discrete change points.

In order their range of estimates be visible, the first plot is on the log scale and the second plot on the linear scale; note the narrower range of years in the right-hand plot resulting from sampling. The posterior mean of s is roughly 1891.

Discrete Sampling

The generated quantities block may be used to draw discrete parameter values using the built-in pseudo-random number generators. For example, with `lp` defined as above, the following program draws a random value for `s` at every iteration.

```
generated quantities {
  int<lower=1,upper=T> s;
  s = categorical_logit_rng(lp);
}
```

A posterior histogram of draws for s is shown on the second change point posterior figure above.

Compared to working in terms of expectations, discrete sampling is highly inefficient, especially for tails of distributions, so this approach should only be used if draws from

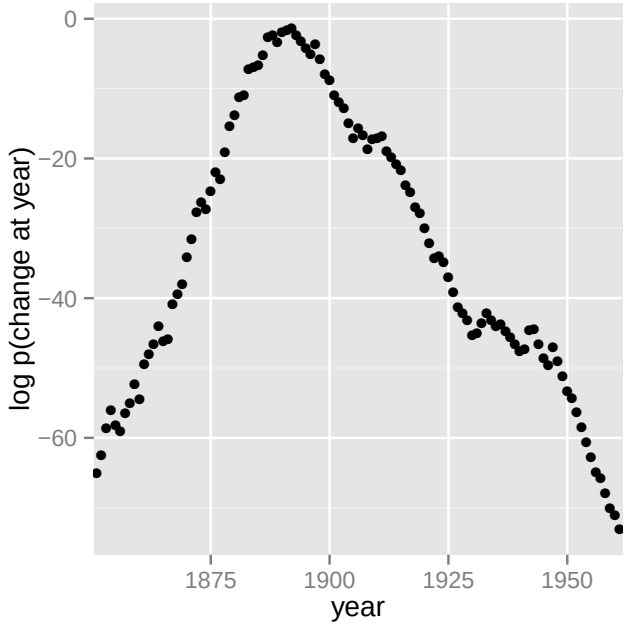

Figure 7.1: Analytical change-point posterior

a distribution are explicitly required. Otherwise, expectations should be computed in the generated quantities block based on the posterior distribution for `s` given by `softmax(lp)`.

Posterior Covariance

The discrete sample generated for s can be used to calculate covariance with other parameters. Although the sampling approach is straightforward, it is more statistically efficient (in the sense of requiring far fewer iterations for the same degree of accuracy) to calculate these covariances in expectation using `lp`.

Multiple Change Points

There is no obstacle in principle to allowing multiple change points. The only issue is that computation increases from linear to quadratic in marginalizing out two change points, cubic for three change points, and so on. There are three parameters, `e`, `m`,

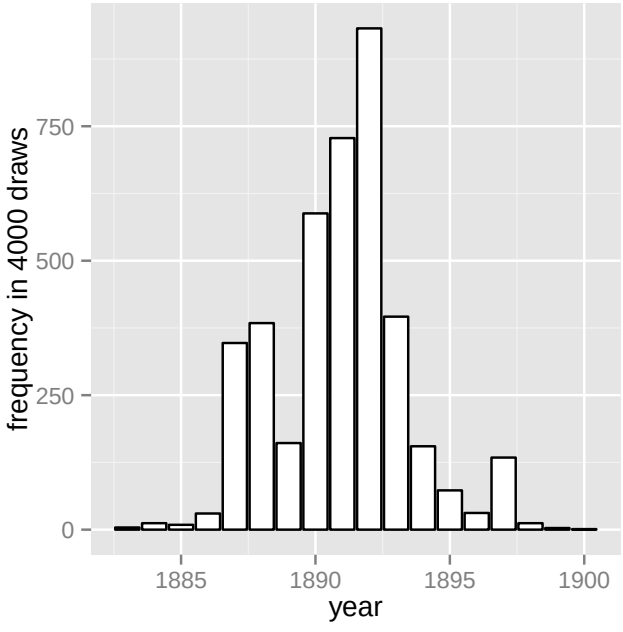

Figure 7.2: Sampled change-point posterior

and `l`, and two loops for the change point and then one over time, with log densities being stored in a matrix.

```
matrix[T, T] lp;
lp = rep_matrix(log_unif, T);
for (s1 in 1:T)
  for (s2 in 1:T)
    for (t in 1:T)
      lp[s1,s2] = lp[s1,s2]
        + poisson_lpmf(D[t] | t < s1 ? e : (t < s2 ? m : l));
```

The matrix can then be converted back to a vector using `to_vector` before being passed to `log_sum_exp`.

7.3. Mark-Recapture Models

A widely applied field method in ecology is to capture (or sight) animals, mark them (e.g., by tagging), then release them. This process is then repeated one or more times, and is often done for populations on an ongoing basis. The resulting data may be used to estimate population size.

The first subsection describes a simple mark-recapture model that does not involve any latent discrete parameters. The following subsections describes the Cormack-Jolly-Seber model, which involves latent discrete parameters for animal death.

Simple Mark-Recapture Model

In the simplest case, a one-stage mark-recapture study produces the following data

- M : number of animals marked in first capture,
- C : number animals in second capture, and
- R : number of marked animals in second capture.

The estimand of interest is

- N : number of animals in the population.

Despite the notation, the model will take N to be a continuous parameter; just because the population must be finite doesn't mean the parameter representing it must be. The parameter will be used to produce a real-valued estimate of the population size.

The Lincoln-Petersen (Lincoln 1930,Petersen (1896)) method for estimating population size is

$$\hat{N} = \frac{MC}{R}.$$

This population estimate would arise from a probabilistic model in which the number of recaptured animals is distributed binomially,

$$R \sim \mathsf{binomial}(C, M/N)$$

given the total number of animals captured in the second round (C) with a recapture probability of M/N, the fraction of the total population N marked in the first round.

```
data {
  int<lower=0> M;
  int<lower=0> C;
  int<lower=0,upper=min(M,C)> R;
```

```
}
parameters {
  real<lower=(C - R + M)> N;
}
model {
  R ~ binomial(C, M / N);
}
```

A probabilistic formulation of the Lincoln-Petersen estimator for population size based on data from a one-step mark-recapture study. The lower bound on N is necessary to efficiently eliminate impossible values.

The probabilistic variant of the Lincoln-Petersen estimator can be directly coded in Stan as shown in the Lincon-Petersen model figure. The Lincoln-Petersen estimate is the maximum likelihood estimate (MLE) for this model.

To ensure the MLE is the Lincoln-Petersen estimate, an improper uniform prior for N is used; this could (and should) be replaced with a more informative prior if possible based on knowledge of the population under study.

The one tricky part of the model is the lower bound $C - R + M$ placed on the population size N. Values below this bound are impossible because it is otherwise not possible to draw R samples out of the C animals recaptured. Implementing this lower bound is necessary to ensure sampling and optimization can be carried out in an unconstrained manner with unbounded support for parameters on the transformed (unconstrained) space. The lower bound in the declaration for C implies a variable transform $f : (C-R+M, \infty) \to (-\infty, +\infty)$ defined by $f(N) = \log(N-(C-R+M))$; the reference manual contains full details of all constrained parameter transforms.

Cormack-Jolly-Seber with Discrete Parameter

The Cormack-Jolly-Seber (CJS) model [Cormack (1964); Jolly:1965; Seber:1965] is an open-population model in which the population may change over time due to death; the presentation here draws heavily on Schofield (2007).

The basic data are

- I: number of individuals,
- T: number of capture periods, and
- $y_{i,t}$: Boolean indicating if individual i was captured at time t.

Each individual is assumed to have been captured at least once because an individual only contributes information conditionally after they have been captured the first

time.

There are two Bernoulli parameters in the model,

- ϕ_t : probability that animal alive at time t survives until $t + 1$ and
- p_t : probability that animal alive at time t is captured at time t.

These parameters will both be given uniform priors, but information should be used to tighten these priors in practice.

The CJS model also employs a latent discrete parameter $z_{i,t}$ indicating for each individual i whether it is alive at time t, distributed as

$$z_{i,t} \sim \text{Bernoulli}(z_{i,t-1} \; ? \; 0 \; : \; \phi_{t-1}).$$

The conditional prevents the model positing zombies; once an animal is dead, it stays dead. The data distribution is then simple to express conditional on z as

$$y_{i,t} \sim \text{Bernoulli}(z_{i,t} \; ? \; 0 \; : \; p_t).$$

The conditional enforces the constraint that dead animals cannot be captured.

Collective Cormack-Jolly-Seber Model

This subsection presents an implementation of the model in terms of counts for different history profiles for individuals over three capture times. It assumes exchangeability of the animals in that each is assigned the same capture and survival probabilities.

In order to ease the marginalization of the latent discrete parameter $z_{i,t}$, the Stan models rely on a derived quantity χ_t for the probability that an individual is never captured again if it is alive at time t (if it is dead, the recapture probability is zero). this quantity is defined recursively by

$$\chi_t = \begin{cases} 1 & \text{if } t = T \\ (1 - \phi_t) + \phi_t(1 - p_{t+1})\chi_{t+1} & \text{if } t < T \end{cases}$$

The base case arises because if an animal was captured in the last time period, the probability it is never captured again is 1 because there are no more capture periods. The recursive case defining χ_t in terms of χ_{t+1} involves two possibilities: (1) not surviving to the next time period, with probability $(1 - \phi_t)$, or (2) surviving to the next time period with probability ϕ_t, not being captured in the next time period with

probability $(1 - p_{t+1})$, and not being captured again after being alive in period $t + 1$ with probability χ_{t+1}.

With three capture times, there are eight captured/not-captured profiles an individual may have. These may be naturally coded as binary numbers as follows.

	captures			
profile	1	2	3	probability
0	−	−	−	n/a
1	−	−	+	n/a
2	−	+	−	χ_2
3	−	+	+	$\phi_2\, p_3$
4	+	−	−	χ_1
5	+	−	+	$\phi_1\,(1 - p_2)\,\phi_2\, p_3$
6	+	+	−	$\phi_1\, p_2\, \chi_2$
7	+	+	+	$\phi_1\, p_2\, \phi_2\, p_3$

History 0, for animals that are never captured, is unobservable because only animals that are captured are observed. History 1, for animals that are only captured in the last round, provides no information for the CJS model, because capture/non-capture status is only informative when conditioned on earlier captures. For the remaining cases, the contribution to the likelihood is provided in the final column.

By defining these probabilities in terms of χ directly, there is no need for a latent binary parameter indicating whether an animal is alive at time t or not. The definition of χ is typically used to define the likelihood (i.e., marginalize out the latent discrete parameter) for the CJS model (Schofield 2007, 9).

The Stan model defines χ as a transformed parameter based on parameters ϕ and p. In the model block, the log probability is incremented for each history based on its count. This second step is similar to collecting Bernoulli observations into a binomial or categorical observations into a multinomial, only it is coded directly in the Stan program using `target +=` rather than being part of a built-in probability function.

The following is the Stan program for the Cormack-Jolly-Seber mark-recapture model that considers counts of individuals with observation histories of being observed or not in three capture periods

```
data {
  int<lower=0> history[7];
```

```
}
parameters {
  real<lower=0,upper=1> phi[2];
  real<lower=0,upper=1> p[3];
}
transformed parameters {
  real<lower=0,upper=1> chi[2];
  chi[2] = (1 - phi[2]) + phi[2] * (1 - p[3]);
  chi[1] = (1 - phi[1]) + phi[1] * (1 - p[2]) * chi[2];
}
model {
  target += history[2] * log(chi[2]);
  target += history[3] * (log(phi[2]) + log(p[3]));
  target += history[4] * (log(chi[1]));
  target += history[5] * (log(phi[1]) + log1m(p[2])
                          + log(phi[2]) + log(p[3]));
  target += history[6] * (log(phi[1]) + log(p[2])
                          + log(chi[2]));
  target += history[7] * (log(phi[1]) + log(p[2])
                          + log(phi[2]) + log(p[3]));
}
generated quantities {
  real<lower=0,upper=1> beta3;
  beta3 = phi[2] * p[3];
}
```

Identifiability

The parameters ϕ_2 and p_3, the probability of death at time 2 and probability of capture at time 3 are not identifiable, because both may be used to account for lack of capture at time 3. Their product, $\beta_3 = \phi_2 \, p_3$, is identified. The Stan model defines `beta3` as a generated quantity. Unidentified parameters pose a problem for Stan's samplers' adaptation. Although the problem posed for adaptation is mild here because the parameters are bounded and thus have proper uniform priors, it would be better to formulate an identified parameterization. One way to do this would be to formulate a hierarchical model for the p and ϕ parameters.

Individual Cormack-Jolly-Seber Model

This section presents a version of the Cormack-Jolly-Seber (CJS) model cast at the individual level rather than collectively as in the previous subsection. It also extends the model to allow an arbitrary number of time periods. The data will consist of the number T of capture events, the number I of individuals, and a boolean flag $y_{i,t}$ indicating if individual i was observed at time t. In Stan,

```
data {
  int<lower=2> T;
  int<lower=0> I;
  int<lower=0,upper=1> y[I, T];
}
```

The advantages to the individual-level model is that it becomes possible to add individual "random effects" that affect survival or capture probability, as well as to avoid the combinatorics involved in unfolding 2^T observation histories for T capture times.

Utility Functions

The individual CJS model is written involves several function definitions. The first two are used in the transformed data block to compute the first and last time period in which an animal was captured.[4]

```
functions {
  int first_capture(int[] y_i) {
    for (k in 1:size(y_i))
      if (y_i[k])
        return k;
    return 0;
  }
  int last_capture(int[] y_i) {
    for (k_rev in 0:(size(y_i) - 1)) {
      int k;
      k = size(y_i) - k_rev;
      if (y_i[k])
        return k;
```

[4] An alternative would be to compute this on the outside and feed it into the Stan model as preprocessed data. Yet another alternative encoding would be a sparse one recording only the capture events along with their time and identifying the individual captured.

```
    }
    return 0;
  }
  ...
}
```

These two functions are used to define the first and last capture time for each individual in the transformed data block.[5]

```
transformed data {
  int<lower=0,upper=T> first[I];
  int<lower=0,upper=T> last[I];
  vector<lower=0,upper=I>[T] n_captured;
  for (i in 1:I)
    first[i] = first_capture(y[i]);
  for (i in 1:I)
    last[i] = last_capture(y[i]);
  n_captured = rep_vector(0, T);
  for (t in 1:T)
    for (i in 1:I)
      if (y[i, t])
        n_captured[t] = n_captured[t] + 1;
}
```

The transformed data block also defines `n_captured[t]`, which is the total number of captures at time `t`. The variable `n_captured` is defined as a vector instead of an integer array so that it can be used in an elementwise vector operation in the generated quantities block to model the population estimates at each time point.

The parameters and transformed parameters are as before, but now there is a function definition for computing the entire vector `chi`, the probability that if an individual is alive at `t` that it will never be captured again.

```
parameters {
  vector<lower=0,upper=1>[T-1] phi;
  vector<lower=0,upper=1>[T] p;
}
transformed parameters {
```

[5]Both functions return 0 if the individual represented by the input array was never captured. Individuals with no captures are not relevant for estimating the model because all probability statements are conditional on earlier captures. Typically they would be removed from the data, but the program allows them to be included even though they make not contribution to the log probability function.

```
  vector<lower=0,upper=1>[T] chi;
  chi = prob_uncaptured(T,p,phi);
}
```

The definition of `prob_uncaptured`, from the functions block, is

```
functions {
  ...
  vector prob_uncaptured(int T, vector p, vector phi) {
    vector[T] chi;
    chi[T] = 1.0;
    for (t in 1:(T - 1)) {
      int t_curr;
      int t_next;
      t_curr = T - t;
      t_next = t_curr + 1;
      chi[t_curr] = (1 - phi[t_curr])
                    + phi[t_curr]
                      * (1 - p[t_next])
                      * chi[t_next];
    }
    return chi;
  }
}
```

The function definition directly follows the mathematical definition of χ_t, unrolling the recursion into an iteration and defining the elements of `chi` from `T` down to 1.

The Model

Given the precomputed quantities, the model block directly encodes the CJS model's log likelihood function. All parameters are left with their default uniform priors and the model simply encodes the log probability of the observations `q` given the parameters `p` and `phi` as well as the transformed parameter `chi` defined in terms of `p` and `phi`.

```
model {
  for (i in 1:I) {
    if (first[i] > 0) {
      for (t in (first[i]+1):last[i]) {
        1 ~ bernoulli(phi[t-1]);
```

```
        y[i, t] ~ bernoulli(p[t]);
      }
      1 ~ bernoulli(chi[last[i]]);
    }
  }
}
```

The outer loop is over individuals, conditional skipping individuals `i` which are never captured. The never-captured check depends on the convention of the first-capture and last-capture functions returning 0 for `first` if an individual is never captured.

The inner loop for individual `i` first increments the log probability based on the survival of the individual with probability `phi[t-1]`. The outcome of 1 is fixed because the individual must survive between the first and last capture (i.e., no zombies). The loop starts after the first capture, because all information in the CJS model is conditional on the first capture.

In the inner loop, the observed capture status `y[i, t]` for individual `i` at time `t` has a Bernoulli distribution based on the capture probability `p[t]` at time `t`.

After the inner loop, the probability of an animal never being seen again after being observed at time `last[i]` is included, because `last[i]` was defined to be the last time period in which animal `i` was observed.

Identified Parameters

As with the collective model described in the previous subsection, this model does not identify `phi[T-1]` and `p[T]`, but does identify their product, `beta`. Thus `beta` is defined as a generated quantity to monitor convergence and report.

```
generated quantities {
  real beta;
  ...

  beta = phi[T-1] * p[T];
  ...
}
```

The parameter `p[1]` is also not modeled and will just be uniform between 0 and 1. A more finely articulated model might have a hierarchical or time-series component, in which case `p[1]` would be an unknown initial condition and both `phi[T-1]` and `p[T]` could be identified.

Population Size Estimates

The generated quantities also calculates an estimate of the population mean at each time `t` in the same way as in the simple mark-recapture model as the number of individuals captured at time `t` divided by the probability of capture at time `t`. This is done with the elementwise division operation for vectors (`./`) in the generated quantities block.

```
generated quantities {
  ...
  vector<lower=0>[T] pop;
  ...
  pop = n_captured ./ p;
  pop[1] = -1;
}
```

Generalizing to Individual Effects

All individuals are modeled as having the same capture probability, but this model could be easily generalized to use a logistic regression here based on individual-level inputs to be used as predictors.

7.4. Data Coding and Diagnostic Accuracy Models

Although seemingly disparate tasks, the rating/coding/annotation of items with categories and diagnostic testing for disease or other conditions share several characteristics which allow their statistical properties to modeled similarly.

Diagnostic Accuracy

Suppose you have diagnostic tests for a condition of varying sensitivity and specificity. Sensitivity is the probability a test returns positive when the patient has the condition and specificity is the probability that a test returns negative when the patient does not have the condition. For example, mammograms and puncture biopsy tests both test for the presence of breast cancer. Mammograms have high sensitivity and low specificity, meaning lots of false positives, whereas puncture biopsies are the opposite, with low sensitivity and high specificity, meaning lots of false negatives.

There are several estimands of interest in such studies. An epidemiological study may be interested in the prevalence of a kind of infection, such as malaria, in a population. A test development study might be interested in the diagnostic accuracy

of a new test. A health care worker performing tests might be interested in the disease status of a particular patient.

Data Coding

Humans are often given the task of coding (equivalently rating or annotating) data. For example, journal or grant reviewers rate submissions, a political study may code campaign commercials as to whether they are attack ads or not, a natural language processing study might annotate Tweets as to whether they are positive or negative in overall sentiment, or a dentist looking at an X-ray classifies a patient as having a cavity or not. In all of these cases, the data coders play the role of the diagnostic tests and all of the same estimands are in play — data coder accuracy and bias, true categories of items being coded, or the prevalence of various categories of items in the data.

Noisy Categorical Measurement Model

In this section, only categorical ratings are considered, and the challenge in the modeling for Stan is to marginalize out the discrete parameters.

Dawid and Skene (1979) introduce a noisy-measurement model for coding and apply it in the epidemiological setting of coding what doctors say about patient histories; the same model can be used for diagnostic procedures.

Data

The data for the model consists of J raters (diagnostic tests), I items (patients), and K categories (condition statuses) to annotate, with $y_{i,j} \in \{1, \ldots, K\}$ being the rating provided by rater j for item i. In a diagnostic test setting for a particular condition, the raters are diagnostic procedures and often $K = 2$, with values signaling the presence or absence of the condition.[6]

It is relatively straightforward to extend Dawid and Skene's model to deal with the situation where not every rater rates each item exactly once.

Model Parameters

The model is based on three parameters, the first of which is discrete:

[6]Diagnostic procedures are often ordinal, as in stages of cancer in oncological diagnosis or the severity of a cavity in dental diagnosis. Dawid and Skene's model may be used as is or naturally generalized for ordinal ratings using a latent continuous rating and cutpoints as in ordinal logistic regression.

- z_i : a value in $\{1, \ldots, K\}$ indicating the true category of item i,
- π : a K-simplex for the prevalence of the K categories in the population, and
- $\theta_{j,k}$: a K-simplex for the response of annotator j to an item of true category k.

Noisy Measurement Model

The true category of an item is assumed to be generated by a simple categorical distribution based on item prevalence,

$$z_i \sim \mathsf{categorical}(\pi).$$

The rating $y_{i,j}$ provided for item i by rater j is modeled as a categorical response of rater i to an item of category z_i,[7]

$$y_{i,j} \sim \mathsf{categorical}(\theta_{j,\pi_{z[i]}}).$$

Priors and Hierarchical Modeling

Dawid and Skene provided maximum likelihood estimates for θ and π, which allows them to generate probability estimates for each z_i.

To mimic Dawid and Skene's maximum likelihood model, the parameters $\theta_{j,k}$ and π can be given uniform priors over K-simplexes. It is straightforward to generalize to Dirichlet priors,

$$\pi \sim \mathsf{Dirichlet}(\alpha)$$

and

$$\theta_{j,k} \sim \mathsf{Dirichlet}(\beta_k)$$

with fixed hyperparameters α (a vector) and β (a matrix or array of vectors). The prior for $\theta_{j,k}$ must be allowed to vary in k, so that, for instance, $\beta_{k,k}$ is large enough to allow the prior to favor better-than-chance annotators over random or adversarial ones.

Because there are J coders, it would be natural to extend the model to include a hierarchical prior for β and to partially pool the estimates of coder accuracy and bias.

[7] In the subscript, $z[i]$ is written as z_i to improve legibility.

Marginalizing out the True Category

Because the true category parameter z is discrete, it must be marginalized out of the joint posterior in order to carry out sampling or maximum likelihood estimation in Stan. The joint posterior factors as

$$p(y, \theta, \pi) = p(y \mid \theta, \pi)\, p(\pi)\, p(\theta),$$

where $p(y \mid \theta, \pi)$ is derived by marginalizing z out of

$$p(z, y \mid \theta, \pi) = \prod_{i=1}^{I} \left(\mathsf{categorical}(z_i \mid \pi) \prod_{j=1}^{J} \mathsf{categorical}(y_{i,j} \mid \theta_{j,z[i]}) \right).$$

This can be done item by item, with

$$p(y \mid \theta, \pi) = \prod_{i=1}^{I} \sum_{k=1}^{K} \left(\mathsf{categorical}(z_i \mid \pi) \prod_{j=1}^{J} \mathsf{categorical}(y_{i,j} \mid \theta_{j,z[i]}) \right).$$

In the missing data model, only the observed labels would be used in the inner product.

Dawid and Skene (1979) derive exactly the same equation in their Equation (2.7), required for the E-step in their expectation maximization (EM) algorithm. Stan requires the marginalized probability function on the log scale,

$$\log p(y \mid \theta, \pi) = \sum_{i=1}^{I} \log \left(\sum_{k=1}^{K} \exp \left(\log \mathsf{categorical}(z_i \mid \pi) \; + \; \sum_{j=1}^{J} \log \mathsf{categorical}(y_{i,j} \mid \theta_{j,z[i]}) \right) \right),$$

which can be directly coded using Stan's built-in `log_sum_exp` function.

Stan Implementation

The Stan program for the Dawid and Skene model is provided below Dawid and Skene (1979).

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

```
  int<lower=1> I;
  int<lower=1> J;

  int<lower=1,upper=K> y[I, J];

  vector<lower=0>[K] alpha;
  vector<lower=0>[K] beta[K];
}
parameters {
  simplex[K] pi;
  simplex[K] theta[J, K];
}
transformed parameters {
  vector[K] log_q_z[I];
  for (i in 1:I) {
    log_q_z[i] = log(pi);
    for (j in 1:J)
      for (k in 1:K)
        log_q_z[i, k] = log_q_z[i, k]
                        + log(theta[j, k, y[i, j]]);
  }
}
model {
  pi ~ dirichlet(alpha);
  for (j in 1:J)
    for (k in 1:K)
      theta[j, k] ~ dirichlet(beta[k]);

  for (i in 1:I)
    target += log_sum_exp(log_q_z[i]);
}
```

The model marginalizes out the discrete parameter z, storing the unnormalized conditional probability $\log q(z_i = k|\theta, \pi)$ in `log_q_z[i, k]`.

The Stan model converges quickly and mixes well using NUTS starting at diffuse initial points, unlike the equivalent model implemented with Gibbs sampling over the discrete parameter. Reasonable weakly informative priors are $\alpha_k = 3$ and $\beta_{k,k} = 2.5K$ and $\beta_{k,k'} = 1$ if $k \neq k'$. Taking α and β_k to be unit vectors and applying

optimization will produce the same answer as the expectation maximization (EM) algorithm of Dawid and Skene (1979).

Inference for the True Category

The quantity `log_q_z[i]` is defined as a transformed parameter. It encodes the (unnormalized) log of $p(z_i \mid \theta, \pi)$. Each iteration provides a value conditioned on that iteration's values for θ and π. Applying the softmax function to `log_q_z[i]` provides a simplex corresponding to the probability mass function of z_i in the posterior. These may be averaged across the iterations to provide the posterior probability distribution over each z_i.

8. Sparse and Ragged Data Structures

Stan does not directly support either sparse or ragged data structures, though both can be accommodated with some programming effort. The sparse matrices chapter introduces a special-purpose sparse matrix times dense vector multiplication, which should be used where applicable; this chapter covers more general data structures.

8.1. Sparse Data Structures

Coding sparse data structures is as easy as moving from a matrix-like data structure to a database-like data structure. For example, consider the coding of sparse data for the IRT models discussed in the item-response model section. There are J students and K questions, and if every student answers every question, then it is practical to declare the data as a $J \times K$ array of answers.

```
data {
  int<lower=1> J;
  int<lower=1> K;
  int<lower=0,upper=1> y[J, K];
  ...
model {
  for (j in 1:J)
    for (k in 1:K)
      y[j, k] ~ bernoulli_logit(delta[k] * (alpha[j] - beta[k]));
  ...
```

On the left is a definition of a sparse matrix y using the NA notation from R (which is

$$y = \begin{bmatrix} 0 & 1 & \text{NA} & 1 \\ 0 & \text{NA} & \text{NA} & 1 \\ \text{NA} & 0 & \text{NA} & \text{NA} \end{bmatrix}$$

jj	kk	y
1	1	0
1	2	1
1	4	1
2	1	0
2	4	1
3	2	0

not supported by Stan). On the right is a database-like encoding of the same sparse matrix y that can be used directly in Stan. The first two columns, jj and kk, denote the indexes and the final column, y, the value. For example, the fifth row of the database-like data structure on the right indicates that $y_{2,4} = 1$.

When not every student is given every question, the dense array coding will no longer work, because Stan does not support undefined values. The sparse data example shows an example with $J = 3$ and $K = 4$, with missing responses shown as NA, as in R. There is no support within Stan for R's NA values, so this data structure cannot be used directly. Instead, it must be converted to a "long form" as in a database, with columns indicating the j and k indexes along with the value. For instance, with jj and kk used for the indexes (following Gelman and Hill (2007)), the data structure can be coded as in the right-hand example in the example. This says that $y_{1,1} = 0$, $y_{1,2} = 1$, and so on, up to $y_{3,2} = 1$, with all other entries undefined.

Letting N be the number of y that are defined, here $N = 6$, the data and model can be formulated as follows.

```
data {
  ...
  int<lower=1> N;
  int<lower=1,upper=J> jj[N];
  int<lower=1,upper=K> kk[N];
  int<lower=0,upper=1> y[N];
  ...
model {
  for (n in 1:N)
    y[n] ~ bernoulli_logit(delta[kk[n]]
                           * (alpha[jj[n]] - beta[kk[n]]));
  ...
```

In the situation where there are no missing values, the two model formulations produce exactly the same log posterior density.

8.2. Ragged Data Structures

Ragged arrays are arrays that are not rectangular, but have different sized entries. This kind of structure crops up when there are different numbers of observations per entry.

A general approach to dealing with ragged structure is to move to a full database-like data structure as discussed in the previous section. A more compact approach is

possible with some indexing into a linear array.

For example, consider a data structure for three groups, each of which has a different number of observations.

$$y_1 = [1.3 \;\; 2.4 \;\; 0.9]$$
$$y_2 = [-1.8 \;\; -0.1]$$
$$y_3 = [12.9 \;\; 18.7 \;\; 42.9 \;\; 4.7]$$

$$z = [1.3 \;\; 2.4 \;\; 0.9 \;\; -1.8 \;\; -0.1 \;\; 12.9 \;\; 18.7 \;\; 42.9 \;\; 4.7]$$
$$s = \{3 \;\; 2 \;\; 4\}$$

On the left is the definition of a ragged data structure y with three rows of different sizes (y_1 is size 3, y_2 size 2, and y_3 size 4). On the right is an example of how to code the data in Stan, using a single vector y to hold all the values and a separate array of integers s to hold the group row sizes. In this example, $y_1 = z_{1:3}$, $y_2 = z_{4:5}$, and $y_3 = z_{6:9}$.

Suppose the model is a simple varying intercept model, which, using vectorized notation, would yield a log-likelihood

$$\sum_{n=1}^{3} \log \mathsf{normal}(y_n \mid \mu_n, \sigma).$$

There's no direct way to encode this in Stan.

A full database type structure could be used, as in the sparse example, but this is inefficient, wasting space for unnecessary indices and not allowing vector-based density operations. A better way to code this data is as a single list of values, with a separate data structure indicating the sizes of each subarray. This is indicated on the right of the example. This coding uses a single array for the values and a separate array for the sizes of each row.

The model can then be coded up using slicing operations as follows.

```
data {
  int<lower=0> N;   // # observations
  int<lower=0> K;   // # of groups
  vector[N] y;      // observations
  int s[K];         // group sizes
  ...
model {
  int pos;
  pos = 1;
  for (k in 1:K) {
    segment(y, pos, s[k]) ~ normal(mu[k], sigma);
    pos = pos + s[k];
```

```
}
```

This coding allows for efficient vectorization, which is worth the copy cost entailed by the `segment()` vector slicing operation.

9. Clustering Models

Unsupervised methods for organizing data into groups are collectively referred to as clustering. This chapter describes the implementation in Stan of two widely used statistical clustering models, soft K-means and latent Dirichlet allocation (LDA). In addition, this chapter includes naive Bayesian classification, which can be viewed as a form of clustering which may be supervised. These models are typically expressed using discrete parameters for cluster assignments. Nevertheless, they can be implemented in Stan like any other mixture model by marginalizing out the discrete parameters (see the mixture modeling chapter).

9.1. Relation to Finite Mixture Models

As mentioned in the clustering section, clustering models and finite mixture models are really just two sides of the same coin. The "soft" K-means model described in the next section is a normal mixture model (with varying assumptions about covariance in higher dimensions leading to variants of K-means). Latent Dirichlet allocation is a mixed-membership multinomial mixture.

9.2. Soft *K*-Means

K-means clustering is a method of clustering data represented as D-dimensional vectors. Specifically, there will be N items to be clustered, each represented as a vector $y_n \in \mathbb{R}^D$. In the "soft" version of K-means, the assignments to clusters will be probabilistic.

Geometric Hard *K*-Means Clustering

K-means clustering is typically described geometrically in terms of the following algorithm, which assumes the number of clusters K and data vectors y as input.

1. For each n in $\{1, \ldots, N\}$, randomly assign vector y_n to a cluster in $\{1, \ldots, K\}$;
2. Repeat
    1. For each cluster k in $\{1, \ldots, K\}$, compute the cluster centroid μ_k by averaging the vectors assigned to that cluster;
    2. For each n in $\{1, \ldots, N\}$, reassign y_n to the cluster k for which the (Euclidean) distance from y_n to μ_k is smallest;
    3. If no vectors changed cluster, return the cluster assignments.

This algorithm is guaranteed to terminate.

Soft *K*-Means Clustering

Soft K-means clustering treats the cluster assignments as probability distributions over the clusters. Because of the connection between Euclidean distance and multivariate normal models with a fixed covariance, soft K-means can be expressed (and coded in Stan) as a multivariate normal mixture model.

In the full generative model, each data point n in $\{1, \ldots, N\}$ is assigned a cluster $z_n \in \{1, \ldots, K\}$ with symmetric uniform probability,

$$z_n \sim \mathsf{categorical}(1/K),$$

where 1 is the unit vector of K dimensions, so that $1/K$ is the symmetric K-simplex. Thus the model assumes that each data point is drawn from a hard decision about cluster membership. The softness arises only from the uncertainty about which cluster generated a data point.

The data points themselves are generated from a multivariate normal distribution whose parameters are determined by the cluster assignment z_n,

$$y_n \sim \mathsf{normal}(\mu_{z[n]}, \Sigma_{z[n]})$$

The sample implementation in this section assumes a fixed unit covariance matrix shared by all clusters k,

$$\Sigma_k = \text{diag\_matrix}(\mathbf{1}),$$

so that the log multivariate normal can be implemented directly up to a proportion by

$$\text{normal}\left(y_n | \mu_k, \text{diag\_matrix}(\mathbf{1})\right) \propto \exp\left(-\frac{1}{2}\sum_{d=1}^{D}\left(\mu_{k,d} - y_{n,d}\right)^2\right).$$

The spatial perspective on K-means arises by noting that the inner term is just half the negative Euclidean distance from the cluster mean μ_k to the data point y_n.

Stan Implementation of Soft *K*-Means

Consider the following Stan program for implementing K-means clustering.

```
data {
  int<lower=0> N;  // number of data points
  int<lower=1> D;  // number of dimensions
```

```
  int<lower=1> K;  // number of clusters
  vector[D] y[N];  // observations
}
transformed data {
  real<upper=0> neg_log_K;
  neg_log_K = -log(K);
}
parameters {
  vector[D] mu[K]; // cluster means
}
transformed parameters {
  real<upper=0> soft_z[N, K]; // log unnormalized clusters
  for (n in 1:N)
    for (k in 1:K)
      soft_z[n, k] = neg_log_K
                     - 0.5 * dot_self(mu[k] - y[n]);
}
model {
  // prior
  for (k in 1:K)
    mu[k] ~ std_normal();

  // likelihood
  for (n in 1:N)
    target += log_sum_exp(soft_z[n]);
}
```

There is an independent standard normal prior on the centroid parameters; this prior could be swapped with other priors, or even a hierarchical model to fit an overall problem scale and location.

The only parameter is `mu`, where `mu[k]` is the centroid for cluster k. The transformed parameters `soft_z[n]` contain the log of the unnormalized cluster assignment probabilities. The vector `soft_z[n]` can be converted back to a normalized simplex using the softmax function (see the functions reference manual), either externally or within the model's generated quantities block.

Generalizing Soft *K*-Means

The multivariate normal distribution with unit covariance matrix produces a log probability density proportional to Euclidean distance (i.e., L_2 distance). Other

distributions relate to other geometries. For instance, replacing the normal distribution with the double exponential (Laplace) distribution produces a clustering model based on L_1 distance (i.e., Manhattan or taxicab distance).

Within the multivariate normal version of K-means, replacing the unit covariance matrix with a shared covariance matrix amounts to working with distances defined in a space transformed by the inverse covariance matrix.

Although there is no global spatial analog, it is common to see soft K-means specified with a per-cluster covariance matrix. In this situation, a hierarchical prior may be used for the covariance matrices.

9.3. The Difficulty of Bayesian Inference for Clustering

Two problems make it pretty much impossible to perform full Bayesian inference for clustering models, the lack of parameter identifiability and the extreme multimodality of the posteriors. There is additional discussion related to the non-identifiability due to label switching in the label switching section.

Non-Identifiability

Cluster assignments are not identified—permuting the cluster mean vectors `mu` leads to a model with identical likelihoods. For instance, permuting the first two indexes in `mu` and the first two indexes in each `soft_z[n]` leads to an identical likelihood (and prior).

The lack of identifiability means that the cluster parameters cannot be compared across multiple Markov chains. In fact, the only parameter in soft K-means is not identified, leading to problems in monitoring convergence. Clusters can even fail to be identified within a single chain, with indices swapping if the chain is long enough or the data are not cleanly separated.

Multimodality

The other problem with clustering models is that their posteriors are highly multimodal. One form of multimodality is the non-identifiability leading to index swapping. But even without the index problems the posteriors are highly multimodal.

Bayesian inference fails in cases of high multimodality because there is no way to visit all of the modes in the posterior in appropriate proportions and thus no way to evaluate integrals involved in posterior predictive inference.

In light of these two problems, the advice often given in fitting clustering models is to try many different initializations and select the sample with the highest overall probability. It is also popular to use optimization-based point estimators such as expectation maximization or variational Bayes, which can be much more efficient than sampling-based approaches.

9.4. Naive Bayes Classification and Clustering

Naive Bayes is a kind of mixture model that can be used for classification or for clustering (or a mix of both), depending on which labels for items are observed.[1]

Multinomial mixture models are referred to as "naive Bayes" because they are often applied to classification problems where the multinomial independence assumptions are clearly false.

Naive Bayes classification and clustering can be applied to any data with multinomial structure. A typical example of this is natural language text classification and clustering, which is used an example in what follows.

The observed data consists of a sequence of M documents made up of bags of words drawn from a vocabulary of V distinct words. A document m has N_m words, which are indexed as $w_{m,1}, \ldots, w_{m,N[m]} \in \{1, \ldots, V\}$. Despite the ordered indexing of words in a document, this order is not part of the model, which is clearly defective for natural human language data. A number of topics (or categories) K is fixed.

The multinomial mixture model generates a single category $z_m \in \{1, \ldots, K\}$ for each document $m \in \{1, \ldots, M\}$ according to a categorical distribution,

$$z_m \sim \mathsf{categorical}(\theta).$$

The K-simplex parameter θ represents the prevalence of each category in the data.

Next, the words in each document are generated conditionally independently of each other and the words in other documents based on the category of the document, with word n of document m being generated as

$$w_{m,n} \sim \mathsf{categorical}(\phi_{z[m]}).$$

The parameter $\phi_{z[m]}$ is a V-simplex representing the probability of each word in the vocabulary in documents of category z_m.

[1] For clustering, the non-identifiability problems for all mixture models present a problem, whereas there is no such problem for classification. Despite the difficulties with full Bayesian inference for clustering, researchers continue to use it, often in an exploratory data analysis setting rather than for predictive modeling.

The parameters θ and ϕ are typically given symmetric Dirichlet priors. The prevalence θ is sometimes fixed to produce equal probabilities for each category $k \in \{1, \ldots, K\}$.

Coding Ragged Arrays

The specification for naive Bayes in the previous sections have used a ragged array notation for the words w. Because Stan does not support ragged arrays, the models are coded using an alternative strategy that provides an index for each word in a global list of words. The data is organized as follows, with the word arrays laid out in a column and each assigned to its document in a second column.

n	w[n]	doc[n]
1	$w_{1,1}$	1
2	$w_{1,2}$	1
$\vdots$	$\vdots$	$\vdots$
N_1	$w_{1,N[1]}$	1
$N_1 + 1$	$w_{2,1}$	2
$N_1 + 2$	$w_{2,2}$	2
$\vdots$	$\vdots$	$\vdots$
$N_1 + N_2$	$w_{2,N[2]}$	2
$N_1 + N_2 + 1$	$w_{3,1}$	3
$\vdots$	$\vdots$	$\vdots$
$N = \sum_{m=1}^{M} N_m$	$w_{M,N[M]}$	M

The relevant variables for the program are `N`, the total number of words in all the documents, the word array `w`, and the document identity array `doc`.

Estimation with Category-Labeled Training Data

A naive Bayes model for estimating the simplex parameters given training data with documents of known categories can be coded in Stan as follows

```
data {
  // training data
  int<lower=1> K;               // num topics
  int<lower=1> V;               // num words
  int<lower=0> M;               // num docs
```

```
  int<lower=0> N;                // total word instances
  int<lower=1,upper=K> z[M];     // topic for doc m
  int<lower=1,upper=V> w[N];     // word n
  int<lower=1,upper=M> doc[N];   // doc ID for word n
  // hyperparameters
  vector<lower=0>[K] alpha;      // topic prior
  vector<lower=0>[V] beta;       // word prior
}
parameters {
  simplex[K] theta;              // topic prevalence
  simplex[V] phi[K];             // word dist for topic k
}
model {
  theta ~ dirichlet(alpha);
  for (k in 1:K)
    phi[k] ~ dirichlet(beta);
  for (m in 1:M)
    z[m] ~ categorical(theta);
  for (n in 1:N)
    w[n] ~ categorical(phi[z[doc[n]]]);
}
```

The topic identifiers z_m are declared as data and the latent category assignments are included as part of the likelihood function.

Estimation without Category-Labeled Training Data

Naive Bayes models can be used in an unsupervised fashion to cluster multinomial-structured data into a fixed number K of categories. The data declaration includes the same variables as the model in the previous section excluding the topic labels z. Because z is discrete, it needs to be summed out of the model calculation. This is done for naive Bayes as for other mixture models. The parameters are the same up to the priors, but the likelihood is now computed as the marginal document probability

$$\begin{aligned}
&\log p(w_{m,1},\ldots,w_{m,N_m} \mid \theta,\phi) \\
&\quad = \log \sum_{k=1}^{K} \left(\mathsf{categorical}(k \mid \theta) \times \prod_{n=1}^{N_m} \mathsf{categorical}(w_{m,n} \mid \phi_k) \right) \\
&\quad = \log \sum_{k=1}^{K} \exp \left(\log \mathsf{categorical}(k \mid \theta) + \sum_{n=1}^{N_m} \log \mathsf{categorical}(w_{m,n} \mid \phi_k) \right).
\end{aligned}$$

The last step shows how the `log_sum_exp` function can be used to stabilize the numerical calculation and return a result on the log scale.

```
model {
  real gamma[M, K];
  theta ~ dirichlet(alpha);
  for (k in 1:K)
    phi[k] ~ dirichlet(beta);
  for (m in 1:M)
    for (k in 1:K)
      gamma[m, k] = categorical_lpmf(k \mid theta);
  for (n in 1:N)
    for (k in 1:K)
      gamma[doc[n], k] = gamma[doc[n], k]
                         + categorical_lpmf(w[n] \mid phi[k]);
  for (m in 1:M)
    target += log_sum_exp(gamma[m]);
}
```

The local variable `gamma[m, k]` represents the value

$$\gamma_{m,k} = \log \mathsf{categorical}(k \mid \theta) + \sum_{n=1}^{N_m} \log \mathsf{categorical}(w_{m,n} \mid \phi_k).$$

Given γ, the posterior probability that document m is assigned category k is

$$\Pr[z_m = k | w, \alpha, \beta] = \exp \left(\gamma_{m,k} - \log \sum_{k=1}^{K} \exp\left(\gamma_{m,k}\right) \right).$$

If the variable `gamma` were declared and defined in the transformed parameter block, its sampled values would be saved by Stan. The normalized posterior probabilities could also be defined as generated quantities.

Full Bayesian Inference for Naive Bayes

Full Bayesian posterior predictive inference for the naive Bayes model can be implemented in Stan by combining the models for labeled and unlabeled data. The estimands include both the model parameters and the posterior distribution over categories for the unlabeled data. The model is essentially a missing data model assuming the unknown category labels are missing completely at random; see Gelman et al. (2013) and Gelman and Hill (2007) for more information on missing data imputation. The model is also an instance of semisupervised learning because the unlabeled data contributes to the parameter estimations.

To specify a Stan model for performing full Bayesian inference, the model for labeled data is combined with the model for unlabeled data. A second document collection is declared as data, but without the category labels, leading to new variables `M2` `N2`, `w2`, and `doc2`. The number of categories and number of words, as well as the hyperparameters are shared and only declared once. Similarly, there is only one set of parameters. Then the model contains a single set of statements for the prior, a set of statements for the labeled data, and a set of statements for the unlabeled data.

Prediction without Model Updates

An alternative to full Bayesian inference involves estimating a model using labeled data, then applying it to unlabeled data without updating the parameter estimates based on the unlabeled data. This behavior can be implemented by moving the definition of `gamma` for the unlabeled documents to the generated quantities block. Because the variables no longer contribute to the log probability, they no longer jointly contribute to the estimation of the model parameters.

9.5. Latent Dirichlet Allocation

Latent Dirichlet allocation (LDA) is a mixed-membership multinomial clustering model Blei, Ng, and Jordan (2003) that generalized naive Bayes. Using the topic and document terminology common in discussions of LDA, each document is modeled as having a mixture of topics, with each word drawn from a topic based on the mixing proportions.

The LDA Model

The basic model assumes each document is generated independently based on fixed hyperparameters. For document m, the first step is to draw a topic distribution

simplex θ_m over the K topics,

$$\theta_m \sim \mathsf{Dirichlet}(\alpha).$$

The prior hyperparameter α is fixed to a K-vector of positive values. Each word in the document is generated independently conditional on the distribution θ_m. First, a topic $z_{m,n} \in \{1, \ldots, K\}$ is drawn for the word based on the document-specific topic-distribution,

$$z_{m,n} \sim \mathsf{categorical}(\theta_m).$$

Finally, the word $w_{m,n}$ is drawn according to the word distribution for topic $z_{m,n}$,

$$w_{m,n} \sim \mathsf{categorical}(\phi_{z[m,n]}).$$

The distributions ϕ_k over words for topic k are also given a Dirichlet prior,

$$\phi_k \sim \mathsf{Dirichlet}(\beta)$$

where β is a fixed V-vector of positive values.

Summing out the Discrete Parameters

Although Stan does not (yet) support discrete sampling, it is possible to calculate the marginal distribution over the continuous parameters by summing out the discrete parameters as in other mixture models. The marginal posterior of the topic and word variables is

$$\begin{aligned} p(\theta, \phi \mid w, \alpha, \beta) &\propto p(\theta \mid \alpha)\, p(\phi \mid \beta)\, p(w \mid \theta, \phi) \\ &= \prod_{m=1}^{M} p(\theta_m \mid \alpha) \times \prod_{k=1}^{K} p(\phi_k \mid \beta) \times \prod_{m=1}^{M} \prod_{n=1}^{M[n]} p(w_{m,n} \mid \theta_m, \phi). \end{aligned}$$

The inner word-probability term is defined by summing out the topic assignments,

$$\begin{aligned} p(w_{m,n} \mid \theta_m, \phi) &= \sum_{z=1}^{K} p(z, w_{m,n} \mid \theta_m, \phi) \\ &= \sum_{z=1}^{K} p(z \mid \theta_m)\, p(w_{m,n} \mid \phi_z). \end{aligned}$$

Plugging the distributions in and converting to the log scale provides a formula that can be implemented directly in Stan,

$$\begin{aligned}
&\log p(\theta, \phi \mid w, \alpha, \beta) \\
&\quad = \sum_{m=1}^{M} \log \mathsf{Dirichlet}(\theta_m \mid \alpha) + \sum_{k=1}^{K} \log \mathsf{Dirichlet}(\phi_k \mid \beta) \\
&\qquad + \sum_{m=1}^{M} \sum_{n=1}^{N[m]} \log \left(\sum_{z=1}^{K} \mathsf{categorical}(z \mid \theta_m) \times \mathsf{categorical}(w_{m,n} \mid \phi_z) \right)
\end{aligned}$$

Implementation of LDA

Applying the marginal derived in the last section to the data structure described in this section leads to the following Stan program for LDA.

```
data {
  int<lower=2> K;               // num topics
  int<lower=2> V;               // num words
  int<lower=1> M;               // num docs
  int<lower=1> N;               // total word instances
  int<lower=1,upper=V> w[N];    // word n
  int<lower=1,upper=M> doc[N];  // doc ID for word n
  vector<lower=0>[K] alpha;     // topic prior
  vector<lower=0>[V] beta;      // word prior
}
parameters {
  simplex[K] theta[M];          // topic dist for doc m
  simplex[V] phi[K];            // word dist for topic k
}
model {
  for (m in 1:M)
    theta[m] ~ dirichlet(alpha);  // prior
  for (k in 1:K)
    phi[k] ~ dirichlet(beta);     // prior
  for (n in 1:N) {
    real gamma[K];
    for (k in 1:K)
      gamma[k] = log(theta[doc[n], k]) + log(phi[k, w[n]]);
```

```
    target += log_sum_exp(gamma);  // likelihood;
  }
}
```

As in the other mixture models, the log-sum-of-exponents function is used to stabilize the numerical arithmetic.

Correlated Topic Model

To account for correlations in the distribution of topics for documents, Blei and Lafferty (2007) introduced a variant of LDA in which the Dirichlet prior on the per-document topic distribution is replaced with a multivariate logistic normal distribution.

The authors treat the prior as a fixed hyperparameter. They use an L_1-regularized estimate of covariance, which is equivalent to the maximum a posteriori estimate given a double-exponential prior. Stan does not (yet) support maximum a posteriori estimation, so the mean and covariance of the multivariate logistic normal must be specified as data.

Fixed Hyperparameter Correlated Topic Model

The Stan model in the previous section can be modified to implement the correlated topic model by replacing the Dirichlet topic prior `alpha` in the data declaration with the mean and covariance of the multivariate logistic normal prior.

```
data {
  ... data as before without alpha ...
  vector[K] mu;           // topic mean
  cov_matrix[K] Sigma;    // topic covariance
}
```

Rather than drawing the simplex parameter `theta` from a Dirichlet, a parameter `eta` is drawn from a multivariate normal distribution and then transformed using softmax into a simplex.

```
parameters {
  simplex[V] phi[K];      // word dist for topic k
  vector[K] eta[M];       // topic dist for doc m
}
transformed parameters {
  simplex[K] theta[M];
```

```
  for (m in 1:M)
    theta[m] = softmax(eta[m]);
}
model {
  for (m in 1:M)
    eta[m] ~ multi_normal(mu, Sigma);
  ... model as before w/o prior for theta ...
}
```

Full Bayes Correlated Topic Model

By adding a prior for the mean and covariance, Stan supports full Bayesian inference for the correlated topic model. This requires moving the declarations of topic mean `mu` and covariance `Sigma` from the data block to the parameters block and providing them with priors in the model. A relatively efficient and interpretable prior for the covariance matrix `Sigma` may be encoded as follows.

```
... data block as before, but without alpha ...
parameters {
  vector[K] mu;              // topic mean
  corr_matrix[K] Omega;      // correlation matrix
  vector<lower=0>[K] sigma;  // scales
  vector[K] eta[M];          // logit topic dist for doc m
  simplex[V] phi[K];         // word dist for topic k
}
transformed parameters {
  ... eta as above ...
  cov_matrix[K] Sigma;       // covariance matrix
  for (m in 1:K)
    Sigma[m, m] = sigma[m] * sigma[m] * Omega[m, m];
  for (m in 1:(K-1)) {
    for (n in (m+1):K) {
      Sigma[m, n] = sigma[m] * sigma[n] * Omega[m, n];
      Sigma[n, m] = Sigma[m, n];
    }
  }
}
model {
  mu ~ normal(0, 5);      // vectorized, diffuse
```

```
  Omega ~ lkj_corr(2.0);  // regularize to unit correlation
  sigma ~ cauchy(0, 5);   // half-Cauchy due to constraint
  ... words sampled as above ...
}
```

The LKJCorr distribution with shape $\alpha > 0$ has support on correlation matrices (i.e., symmetric positive definite with unit diagonal). Its density is defined by

$$\mathsf{LkjCorr}(\Omega|\alpha) \propto \det(\Omega)^{\alpha-1}$$

With a scale of $\alpha = 2$, the weakly informative prior favors a unit correlation matrix. Thus the compound effect of this prior on the covariance matrix Σ for the multivariate logistic normal is a slight concentration around diagonal covariance matrices with scales determined by the prior on `sigma`.

10. Gaussian Processes

Gaussian processes are continuous stochastic processes and thus may be interpreted as providing a probability distribution over functions. A probability distribution over continuous functions may be viewed, roughly, as an uncountably infinite collection of random variables, one for each valid input. The generality of the supported functions makes Gaussian priors popular choices for priors in general multivariate (non-linear) regression problems.

The defining feature of a Gaussian process is that the joint distribution of the function's value at a finite number of input points is a multivariate normal distribution. This makes it tractable to both fit models from finite amounts of observed data and make predictions for finitely many new data points.

Unlike a simple multivariate normal distribution, which is parameterized by a mean vector and covariance matrix, a Gaussian process is parameterized by a mean function and covariance function. The mean and covariance functions apply to vectors of inputs and return a mean vector and covariance matrix which provide the mean and covariance of the outputs corresponding to those input points in the functions drawn from the process.

Gaussian processes can be encoded in Stan by implementing their mean and covariance functions and plugging the result into the Gaussian form of their sampling distribution, or by using the specialized covariance functions outlined below. This form of model is straightforward and may be used for simulation, model fitting, or posterior predictive inference. A more efficient Stan implementation for the GP with a normally distributed outcome marginalizes over the latent Gaussian process, and applies a Cholesky-factor reparameterization of the Gaussian to compute the likelihood and the posterior predictive distribution analytically.

After defining Gaussian processes, this chapter covers the basic implementations for simulation, hyperparameter estimation, and posterior predictive inference for univariate regressions, multivariate regressions, and multivariate logistic regressions. Gaussian processes are general, and by necessity this chapter only touches on some basic models. For more information, see Rasmussen and Williams (2006).

10.1. Gaussian Process Regression

The data for a multivariate Gaussian process regression consists of a series of N inputs $x_1, \ldots, x_N \in \mathbb{R}^D$ paired with outputs $y_1, \ldots, y_N \in \mathbb{R}$. The defining feature of Gaussian processes is that the probability of a finite number of outputs y conditioned on their inputs x is Gaussian:

$$y \sim \text{multivariate normal}(m(x), K(x \mid \theta)),$$

where $m(x)$ is an N-vector and $K(x \mid \theta)$ is an $N \times N$ covariance matrix. The mean function $m : \mathbb{R}^{N \times D} \to \mathbb{R}^N$ can be anything, but the covariance function $K : \mathbb{R}^{N \times D} \to \mathbb{R}^{N \times N}$ must produce a positive-definite matrix for any input x.[1]

A popular covariance function, which will be used in the implementations later in this chapter, is an exponentiated quadratic function,

$$K(x \mid \alpha, \rho, \sigma)_{i,j} = \alpha^2 \exp\left(-\frac{1}{2\rho^2} \sum_{d=1}^{D} (x_{i,d} - x_{j,d})^2\right) + \delta_{i,j}\sigma^2,$$

where α, ρ, and σ are hyperparameters defining the covariance function and where $\delta_{i,j}$ is the Kronecker delta function with value 1 if $i = j$ and value 0 otherwise; this test is between the indexes i and j, not between values x_i and x_j. This kernel is obtained through a convolution of two independent Gaussian processes, f_1 and f_2, with kernels

$$K_1(x \mid \alpha, \rho)_{i,j} = \alpha^2 \exp\left(-\frac{1}{2\rho^2} \sum_{d=1}^{D} (x_{i,d} - x_{j,d})^2\right)$$

and

$$K_2(x \mid \sigma)_{i,j} = \delta_{i,j}\sigma^2,$$

The addition of σ^2 on the diagonal is important to ensure the positive definiteness of the resulting matrix in the case of two identical inputs $x_i = x_j$. In statistical terms, σ is the scale of the noise term in the regression.

The hyperparameter ρ is the *length-scale*, and corresponds to the frequency of the functions represented by the Gaussian process prior with respect to the domain. Values of ρ closer to zero lead the GP to represent high-frequency functions, whereas larger values of ρ lead to low-frequency functions. The hyperparameter α is the

[1] Gaussian processes can be extended to covariance functions producing positive semi-definite matrices, but Stan does not support inference in the resulting models because the resulting distribution does not have unconstrained support.

marginal standard deviation. It controls the magnitude of the range of the function represented by the GP. If you were to take the standard deviation of many draws from the GP f_1 prior at a single input x conditional on one value of α one would recover α.

The only term in the squared exponential covariance function involving the inputs x_i and x_j is their vector difference, $x_i - x_j$. This produces a process with stationary covariance in the sense that if an input vector x is translated by a vector ϵ to $x + \epsilon$, the covariance at any pair of outputs is unchanged, because $K(x \mid \theta) = K(x + \epsilon \mid \theta)$.

The summation involved is just the squared Euclidean distance between x_i and x_j (i.e., the L_2 norm of their difference, $x_i - x_j$). This results in support for smooth functions in the process. The amount of variation in the function is controlled by the free hyperparameters α, ρ, and σ.

Changing the notion of distance from Euclidean to taxicab distance (i.e., an L_1 norm) changes the support to functions which are continuous but not smooth.

10.2. Simulating from a Gaussian Process

It is simplest to start with a Stan model that does nothing more than simulate draws of functions f from a Gaussian process. In practical terms, the model will draw values $y_n = f(x_n)$ for finitely many input points x_n.

The Stan model defines the mean and covariance functions in a transformed data block and then samples outputs y in the model using a multivariate normal distribution. To make the model concrete, the squared exponential covariance function described in the previous section will be used with hyperparameters set to $\alpha^2 = 1$, $\rho^2 = 1$, and $\sigma^2 = 0.1$, and the mean function m is defined to always return the zero vector, $m(x) = \mathbf{0}$. Consider the following implementation of a Gaussian process simulator.

```
data {
  int<lower=1> N;
  real x[N];
}
transformed data {
  matrix[N, N] K;
  vector[N] mu = rep_vector(0, N);
  for (i in 1:(N - 1)) {
    K[i, i] = 1 + 0.1;
    for (j in (i + 1):N) {
```

```
      K[i, j] = exp(-0.5 * square(x[i] - x[j]));
      K[j, i] = K[i, j];
    }
  }
  K[N, N] = 1 + 0.1;
}
parameters {
  vector[N] y;
}
model {
  y ~ multi_normal(mu, K);
}
```

The above model can also be written more compactly using the specialized covariance function that implements the exponentiated quadratic kernel.

```
data {
  int<lower=1> N;
  real x[N];
}
transformed data {
  matrix[N, N] K = cov_exp_quad(x, 1.0, 1.0);
  vector[N] mu = rep_vector(0, N);
  for (n in 1:N)
    K[n, n] = K[n, n] + 0.1;
}
parameters {
  vector[N] y;
}
model {
  y ~ multi_normal(mu, K);
}
```

The input data are just the vector of inputs `x` and its size `N`. Such a model can be used with values of `x` evenly spaced over some interval in order to plot sample draws of functions from a Gaussian process.

Multivariate Inputs

Only the input data needs to change in moving from a univariate model to a multivariate model.

The only lines that change from the univariate model above are as follows.

```
data {
  int<lower=1> N;
  int<lower=1> D;
  vector[D] x[N];
}
transformed data {
...
```

The data are now declared as an array of vectors instead of an array of scalars; the dimensionality `D` is also declared.

In the remainder of the chapter, univariate models will be used for simplicity, but any of the models could be changed to multivariate in the same way as the simple sampling model. The only extra computational overhead from a multivariate model is in the distance calculation.

Cholesky Factored and Transformed Implementation

A more efficient implementation of the simulation model can be coded in Stan by relocating, rescaling and rotating an isotropic standard normal variate. Suppose η is an an isotropic standard normal variate

$$\eta \sim \mathsf{normal}(\mathbf{0}, \mathbf{1}),$$

where $\mathbf{0}$ is an N-vector of 0 values and $\mathbf{1}$ is the $N \times N$ identity matrix. Let L be the Cholesky decomposition of $K(x \mid \theta)$, i.e., the lower-triangular matrix L such that $LL^\top = K(x \mid \theta)$. Then the transformed variable $\mu + L\eta$ has the intended target distribution,

$$\mu + L\eta \sim \mathsf{multivariate\ normal}(\mu(x), K(x \mid \theta)).$$

This transform can be applied directly to Gaussian process simulation.

This model has the same data declarations for `N` and `x`, and the same transformed data definitions of `mu` and `K` as the previous model, with the addition of a transformed data variable for the Cholesky decomposition. The parameters change to the raw parameters sampled from an isotropic standard normal, and the actual samples are defined as generated quantities.

```
...
transformed data {
  matrix[N, N] L;
```

```
...
  L = cholesky_decompose(K);
}
parameters {
  vector[N] eta;
}
model {
  eta ~ std_normal();
}
generated quantities {
  vector[N] y;
  y = mu + L * eta;
}
```

The Cholesky decomposition is only computed once, after the data are loaded and the covariance matrix `K` computed. The isotropic normal distribution for `eta` is specified as a vectorized univariate distribution for efficiency; this specifies that each `eta[n]` has an independent standard normal distribution. The sampled vector `y` is then defined as a generated quantity using a direct encoding of the transform described above.

10.3. Fitting a Gaussian Process

GP with a normal outcome

The full generative model for a GP with a normal outcome, $y \in \mathbb{R}^N$, with inputs $x \in \mathbb{R}^N$, for a finite N:

$$
\begin{aligned}
\rho &\sim \mathsf{InvGamma}(5, 5) \\
\alpha &\sim \mathsf{normal}(0, 1) \\
\sigma &\sim \mathsf{normal}(0, 1) \\
f &\sim \mathsf{multivariate\ normal}\left(0, K(x \mid \alpha, \rho)\right) \\
y_i &\sim \mathsf{normal}(f_i, \sigma)\, \forall i \in \{1, \ldots, N\}
\end{aligned}
$$

With a normal outcome, it is possible to integrate out the Gaussian process f, yielding the more parsimonious model:

$$
\begin{aligned}
\rho &\sim \text{InvGamma}(5, 5) \\
\alpha &\sim \text{normal}(0, 1) \\
\sigma &\sim \text{normal}(0, 1) \\
y &\sim \text{multivariate normal } \left(0, K(x \mid \alpha, \rho) + \mathbf{I}_N \sigma^2\right)
\end{aligned}
$$

It can be more computationally efficient when dealing with a normal outcome to integrate out the Gaussian process, because this yields a lower-dimensional parameter space over which to do inference. We'll fit both models in Stan. The former model will be referred to as the latent variable GP, while the latter will be called the marginal likelihood GP.

The hyperparameters controlling the covariance function of a Gaussian process can be fit by assigning them priors, like we have in the generative models above, and then computing the posterior distribution of the hyperparameters given observed data. The priors on the parameters should be defined based on prior knowledge of the scale of the output values (α), the scale of the output noise (σ), and the scale at which distances are measured among inputs (ρ). See the Gaussian process priors section for more information about how to specify appropriate priors for the hyperparameters.

The Stan program implementing the marginal likelihood GP is shown below. The program is similar to the Stan programs that implement the simulation GPs above, but because we are doing inference on the hyperparameters, we need to calculate the covariance matrix `K` in the model block, rather than the transformed data block.

```
data {
  int<lower=1> N;
  real x[N];
  vector[N] y;
}
transformed data {
  vector[N] mu = rep_vector(0, N);
}
parameters {
  real<lower=0> rho;
  real<lower=0> alpha;
  real<lower=0> sigma;
```

```
}
model {
  matrix[N, N] L_K;
  matrix[N, N] K = cov_exp_quad(x, alpha, rho);
  real sq_sigma = square(sigma);

  // diagonal elements
  for (n in 1:N)
    K[n, n] = K[n, n] + sq_sigma;

  L_K = cholesky_decompose(K);

  rho ~ inv_gamma(5, 5);
  alpha ~ std_normal();
  sigma ~ std_normal();

  y ~ multi_normal_cholesky(mu, L_K);
}
```

The data block now declares a vector `y` of observed values `y[n]` for inputs `x[n]`. The transformed data block now only defines the mean vector to be zero. The three hyperparameters are defined as parameters constrained to be non-negative. The computation of the covariance matrix `K` is now in the model block because it involves unknown parameters and thus can't simply be precomputed as transformed data. The rest of the model consists of the priors for the hyperparameters and the multivariate Cholesky-parameterized normal likelihood, only now the value `y` is known and the covariance matrix `K` is an unknown dependent on the hyperparameters, allowing us to learn the hyperparameters.

We have used the Cholesky parameterized multivariate normal rather than the standard parameterization because it allows us to the `cholesky_decompose` function which has been optimized for both small and large matrices. When working with small matrices the differences in computational speed between the two approaches will not be noticeable, but for larger matrices ($N \gtrsim 100$) the Cholesky decomposition version will be faster.

Hamiltonian Monte Carlo sampling is fast and effective for hyperparameter inference in this model Neal (1997). If the posterior is well-concentrated for the hyperparameters the Stan implementation will fit hyperparameters in models with a few hundred data points in seconds.

Latent variable GP

We can also explicitly code the latent variable formulation of a GP in Stan. This will be useful for when the outcome is not normal. We'll need to add a small positive term, δ to the diagonal of the covariance matrix in order to ensure that our covariance matrix remains positive definite.

```
data {
  int<lower=1> N;
  real x[N];
  vector[N] y;
}
transformed data {
  real delta = 1e-9;
}
parameters {
  real<lower=0> rho;
  real<lower=0> alpha;
  real<lower=0> sigma;
  vector[N] eta;
}
model {
  vector[N] f;
  {
    matrix[N, N] L_K;
    matrix[N, N] K = cov_exp_quad(x, alpha, rho);

    // diagonal elements
    for (n in 1:N)
      K[n, n] = K[n, n] + delta;

    L_K = cholesky_decompose(K);
    f = L_K * eta;
  }

  rho ~ inv_gamma(5, 5);
  alpha ~ std_normal();
  sigma ~ std_normal();
  eta ~ std_normal();
```

```
  y ~ normal(f, sigma);
}
```

Two differences between the latent variable GP and the marginal likelihood GP are worth noting. The first is that we have augmented our parameter block with a new parameter vector of length N called 'eta'. This is used in the model block to generate a multivariate normal vector called f, corresponding to the latent GP. We put a normal$(0, 1)$ prior on `eta` like we did in the Cholesky-parameterized GP in the simulation section. The second difference is that our likelihood is now univariate, though we could code N likelihood terms as one N-dimensional multivariate normal with an identity covariance matrix multiplied by σ^2. However, it is more efficient to use the vectorized statement as shown above.

Discrete outcomes with Gaussian Processes

Gaussian processes can be generalized the same way as standard linear models by introducing a link function. This allows them to be used as discrete data models.

Poisson GP

If we want to model count data, we can remove the σ parameter, and use `poisson_log`, which implements a log link, for our likelihood rather than `normal`. We can also add an overall mean parameter, a, which will account for the marginal expected value for y. We do this because we cannot center count data like we would for normally distributed data.

```
data {
...
  int<lower=0> y[N];
...
}
...
parameters {
  real<lower=0> rho;
  real<lower=0> alpha;
  real a;
  vector[N] eta;
}
model {
...
```

```
  rho ~ inv_gamma(5, 5);
  alpha ~ std_normal();
  a ~ std_normal();
  eta ~ std_normal();

  y ~ poisson_log(a + f);
}
```

Logistic Gaussian Process Regression

For binary classification problems, the observed outputs $z_n \in \{0, 1\}$ are binary. These outputs are modeled using a Gaussian process with (unobserved) outputs y_n through the logistic link,

$$z_n \sim \text{Bernoulli}(\text{logit}^{-1}(y_n)),$$

or in other words,

$$\Pr[z_n = 1] = \text{logit}^{-1}(y_n).$$

We can extend our latent variable GP Stan program to deal with classification problems. Below a is the bias term, which can help account for imbalanced classes in the training data:

```
data {
...
  int<lower=0, upper=1> z[N];
...
}
...
model {
...

  y ~ bernoulli_logit(a + f);
}
```

Automatic Relevance Determination

If we have multivariate inputs $x \in \mathbb{R}^D$, the squared exponential covariance function can be further generalized by fitting a scale parameter ρ_d for each dimension d,

$$k(x \mid \alpha, \vec{\rho}, \sigma)_{i,j} = \alpha^2 \exp\left(-\frac{1}{2}\sum_{d=1}^{D}\frac{1}{\rho_d^2}(x_{i,d} - x_{j,d})^2\right) + \delta_{i,j}\sigma^2.$$

The estimation of ρ was termed "automatic relevance determination" in Neal (1996a), but this is misleading, because the magnitude the scale of the posterior for each ρ_d is dependent on the scaling of the input data along dimension d. Moreover, the scale of the parameters ρ_d measures non-linearity along the d-th dimension, rather than "relevance" Piironen and Vehtari (2016).

A priori, the closer ρ_d is to zero, the more nonlinear the conditional mean in dimension d is. A posteriori, the actual dependencies between x and y play a role. With one covariate x_1 having a linear effect and another covariate x_2 having a nonlinear effect, it is possible that $\rho_1 > \rho_2$ even if the predictive relevance of x_1 is higher (Rasmussen and Williams 2006, 80). The collection of ρ_d (or $1/\rho_d$) parameters can also be modeled hierarchically.

The implementation of automatic relevance determination in Stan is straightforward, though it currently requires the user to directly code the covariance matrix. We'll write a function to generate the Cholesky of the covariance matrix called `L_cov_exp_quad_ARD`.

```
functions {
  matrix L_cov_exp_quad_ARD(vector[] x,
                            real alpha,
                            vector rho,
                            real delta) {
    int N = size(x);
    matrix[N, N] K;
    real sq_alpha = square(alpha);
    for (i in 1:(N-1)) {
      K[i, i] = sq_alpha + delta;
      for (j in (i + 1):N) {
        K[i, j] = sq_alpha
                  * exp(-0.5 * dot_self((x[i] - x[j]) ./ rho));
        K[j, i] = K[i, j];
      }
    }
    K[N, N] = sq_alpha + delta;
    return cholesky_decompose(K);
  }
}
data {
  int<lower=1> N;
  int<lower=1> D;
```

```
  vector[D] x[N];
  vector[N] y;
}
transformed data {
  real delta = 1e-9;
}
parameters {
  vector<lower=0>[D] rho;
  real<lower=0> alpha;
  real<lower=0> sigma;
  vector[N] eta;
}
model {
  vector[N] f;
  {
    matrix[N, N] L_K = L_cov_exp_quad_ARD(x, alpha, rho, delta);
    f = L_K * eta;
  }

  rho ~ inv_gamma(5, 5);
  alpha ~ std_normal();
  sigma ~ std_normal();
  eta ~ std_normal();

  y ~ normal(f, sigma);
}
```

Priors for Gaussian Process Parameters

Formulating priors for GP hyperparameters requires the analyst to consider the inherent statistical properties of a GP, the GP's purpose in the model, and the numerical issues that may arise in Stan when estimating a GP.

Perhaps most importantly, the parameters ρ and α are weakly identified Zhang (2004). The ratio of the two parameters is well-identified, but in practice we put independent priors on the two hyperparameters because these two quantities are more interpretable than their ratio.

Priors for length-scale

GPs are a flexible class of priors and, as such, can represent a wide spectrum of functions. For length scales below the minimum spacing of the covariates the GP likelihood plateaus. Unless regularized by a prior, this flat likelihood induces considerable posterior mass at small length scales where the observation variance drops to zero and the functions supported by the GP being to exactly interpolate between the input data. The resulting posterior not only significantly overfits to the input data, it also becomes hard to accurately sample using Euclidean HMC.

We may wish to put further soft constraints on the length-scale, but these are dependent on how the GP is used in our statistical model.

If our model consists of only the GP, i.e.:

$$
\begin{aligned}
f &\sim \text{multivariate normal}\,(0, K(x \mid \alpha, \rho)) \\
y_i &\sim \text{normal}(f_i, \sigma)\,\forall i \in \{1, \ldots, N\} \\
x &\in \mathbb{R}^{N \times D}, \quad f \in \mathbb{R}^N
\end{aligned}
$$

we likely don't need constraints beyond penalizing small length-scales. We'd like to allow the GP prior to represent both high-frequency and low-frequency functions, so our prior should put non-negligible mass on both sets of functions. In this case, an inverse gamma, `inv_gamma_lpdf` in Stan's language, will work well as it has a sharp left tail that puts negligible mass on infinitesimal length-scales, but a generous right tail, allowing for large length-scales. Inverse gamma priors will avoid infinitesimal length-scales because the density is zero at zero, so the posterior for length-scale will be pushed away from zero. An inverse gamma distribution is one of many zero-avoiding or boundary-avoiding distributions. See **??** for more on boundary-avoiding priors.

If we're using the GP as a component in a larger model that includes an overall mean and fixed effects for the same variables we're using as the domain for the GP, i.e.:

$$
\begin{aligned}
f &\sim \text{multivariate normal}\big(0, K(x \mid \alpha, \rho)\big) \\
y_i &\sim \text{normal}\left(\beta_0 + x_i \beta_{[1:D]} + f_i, \sigma\right)\,\forall i \in \{1, \ldots, N\} \\
x_i^T, \beta_{[1:D]} &\in \mathbb{R}^D, \quad x \in \mathbb{R}^{N \times D}, \quad f \in \mathbb{R}^N
\end{aligned}
$$

we'll likely want to constrain large length-scales as well. A length scale that is larger than the scale of the data yields a GP posterior that is practically linear (with respect to the particular covariate) and increasing the length scale has little impact on the likelihood. This will introduce nonidentifiability in our model, as both the fixed effects and the GP will explain similar variation. In order to limit the amount of overlap between the GP and the linear regression, we should use a prior with a sharper right tail to limit the GP to higher-frequency functions. We can use a generalized inverse Gaussian distribution:

$$f(x \mid a, b, p) = \frac{(a/b)^{p/2}}{2K_p\left(\sqrt{ab}\right)} x^{p-1} \exp\big(-(ax + b/x)/2\big)$$
$$x, a, b \in \mathbb{R}^+, \quad p \in \mathbb{Z}$$

which has an inverse gamma left tail if $p \leq 0$ and an inverse Gaussian right tail. This has not yet been implemented in Stan's math library, but it is possible to implement as a user defined function:

```
functions {
  real generalized_inverse_gaussian_lpdf(real x, int p,
                                         real a, real b) {
    return p * 0.5 * log(a / b)
      - log(2 * modified_bessel_second_kind(p, sqrt(a * b)))
      + (p - 1) * log(x)
      - (a * x + b / x) * 0.5;
 }
}
data {
...
```

If we have high-frequency covariates in our fixed effects, we may wish to further regularize the GP away from high-frequency functions, which means we'll need to penalize smaller length-scales. Luckily, we have a useful way of thinking about how length-scale affects the frequency of the functions supported the GP. If we were to repeatedly draw from a zero-mean GP with a length-scale of ρ in a fixed-domain $[0, T]$, we would get a distribution for the number of times each draw of the GP crossed the zero axis. The expectation of this random variable, the number of zero crossings, is $T/\pi\rho$. You can see that as ρ decreases, the expectation of the number of upcrossings increases as the GP is representing higher-frequency functions. Thus,

this is a good statistic to keep in mind when setting a lower-bound for our prior on length-scale in the presence of high-frequency covariates. However, this statistic is only valid for one-dimensional inputs.

Priors for marginal standard deviation

The parameter α corresponds to how much of the variation is explained by the regression function and has a similar role to the prior variance for linear model weights. This means the prior can be the same as used in linear models, such as a half-t prior on α.

A half-t or half-Gaussian prior on alpha also has the benefit of putting nontrivial prior mass around zero. This allows the GP support the zero functions and allows the possibility that the GP won't contribute to the conditional mean of the total output.

Predictive Inference with a Gaussian Process

Suppose for a given sequence of inputs x that the corresponding outputs y are observed. Given a new sequence of inputs $\tilde{x}$, the posterior predictive distribution of their labels is computed by sampling outputs $\tilde{y}$ according to

$$p\left(\tilde{y} \mid \tilde{x}, x, y\right) \;=\; \frac{p\left(\tilde{y}, y \mid \tilde{x}, x\right)}{p(y \mid x)} \;\propto\; p\left(\tilde{y}, y \mid \tilde{x}, x\right).$$

A direct implementation in Stan defines a model in terms of the joint distribution of the observed y and unobserved $\tilde{y}$.

```
data {
  int<lower=1> N1;
  real x1[N1];
  vector[N1] y1;
  int<lower=1> N2;
  real x2[N2];
}
transformed data {
  real delta = 1e-9;
  int<lower=1> N = N1 + N2;
  real x[N];
  for (n1 in 1:N1) x[n1] = x1[n1];
  for (n2 in 1:N2) x[N1 + n2] = x2[n2];
```

```
}
parameters {
  real<lower=0> rho;
  real<lower=0> alpha;
  real<lower=0> sigma;
  vector[N] eta;
}
transformed parameters {
  vector[N] f;
  {
    matrix[N, N] L_K;
    matrix[N, N] K = cov_exp_quad(x, alpha, rho);

    // diagonal elements
    for (n in 1:N)
      K[n, n] = K[n, n] + delta;

    L_K = cholesky_decompose(K);
    f = L_K * eta;
  }
}
model {
  rho ~ inv_gamma(5, 5);
  alpha ~ std_normal();
  sigma ~ std_normal();
  eta ~ std_normal();

  y1 ~ normal(f[1:N1], sigma);
}
generated quantities {
  vector[N2] y2;
  for (n2 in 1:N2)
    y2[n2] = normal_rng(f[N1 + n2], sigma);
}
```

The input vectors `x1` and `x2` are declared as data, as is the observed output vector `y1`. The unknown output vector `y2`, which corresponds to input vector `x2`, is declared in the generated quantities block and will be sampled when the model is executed.

A transformed data block is used to combine the input vectors `x1` and `x2` into a single

vector x.

The model block declares and defines a local variable for the combined output vector f, which consists of the concatenation of the conditional mean for known outputs y1 and unknown outputs y2. Thus the combined output vector f is aligned with the combined input vector x. All that is left is to define the univariate normal sampling statement for y.

The generated quantities block defines the quantity y2. We generate y2 by sampling N2 univariate normals with each mean corresponding to the appropriate element in f.

Predictive Inference in non-Gaussian GPs

We can do predictive inference in non-Gaussian GPs in much the same way as we do with Gaussian GPs.

Consider the following full model for prediction using logistic Gaussian process regression.

```
data {
  int<lower=1> N1;
  real x1[N1];
  int<lower=0, upper=1> z1[N1];
  int<lower=1> N2;
  real x2[N2];
}
transformed data {
  real delta = 1e-9;
  int<lower=1> N = N1 + N2;
  real x[N];
  for (n1 in 1:N1) x[n1] = x1[n1];
  for (n2 in 1:N2) x[N1 + n2] = x2[n2];
}
parameters {
  real<lower=0> rho;
  real<lower=0> alpha;
  real a;
  vector[N] eta;
}
transformed parameters {
```

```
  vector[N] f;
  {
    matrix[N, N] L_K;
    matrix[N, N] K = cov_exp_quad(x, alpha, rho);

    // diagonal elements
    for (n in 1:N)
      K[n, n] = K[n, n] + delta;

    L_K = cholesky_decompose(K);
    f = L_K * eta;
  }
}
model {
  rho ~ inv_gamma(5, 5);
  alpha ~ std_normal();
  a ~ std_normal();
  eta ~ std_normal();

  z1 ~ bernoulli_logit(a + f[1:N1]);
}
generated quantities {
  int z2[N2];
  for (n2 in 1:N2)
    z2[n2] = bernoulli_logit_rng(a + f[N1 + n2]);
}
```

Analytical Form of Joint Predictive Inference

Bayesian predictive inference for Gaussian processes with Gaussian observations can be sped up by deriving the posterior analytically, then directly sampling from it.

Jumping straight to the result,

$$p\left(\tilde{y} \mid \tilde{x}, y, x\right) = \mathsf{normal}\left(K^{\top}\Sigma^{-1}y,\ \Omega - K^{\top}\Sigma^{-1}K\right),$$

where $\Sigma = K(x \mid \alpha, \rho, \sigma)$ is the result of applying the covariance function to the inputs x with observed outputs y, $\Omega = K(\tilde{x} \mid \alpha, \rho)$ is the result of applying the covariance function to the inputs $\tilde{x}$ for which predictions are to be inferred, and

K is the matrix of covariances between inputs x and $\tilde{x}$, which in the case of the exponentiated quadratic covariance function would be

$$K(x \mid \alpha, \rho)_{i,j} = \eta^2 \exp\left(-\frac{1}{2\rho^2}\sum_{d=1}^{D}\left(x_{i,d} - \tilde{x}_{j,d}\right)^2\right).$$

There is no noise term including σ^2 because the indexes of elements in x and $\tilde{x}$ are never the same.

This Stan code below uses the analytic form of the posterior and provides sampling of the resulting multivariate normal through the Cholesky decomposition. The data declaration is the same as for the latent variable example, but we've defined a function called `gp_pred_rng` which will generate a draw from the posterior predictive mean conditioned on observed data `y1`. The code uses a Cholesky decomposition in triangular solves in order to cut down on the the number of matrix-matrix multiplications when computing the conditional mean and the conditional covariance of $p(\tilde{y})$.

```
functions {
  vector gp_pred_rng(real[] x2,
                     vector y1,
                     real[] x1,
                     real alpha,
                     real rho,
                     real sigma,
                     real delta) {
    int N1 = rows(y1);
    int N2 = size(x2);
    vector[N2] f2;
    {
      matrix[N1, N1] L_K;
      vector[N1] K_div_y1;
      matrix[N1, N2] k_x1_x2;
      matrix[N1, N2] v_pred;
      vector[N2] f2_mu;
      matrix[N2, N2] cov_f2;
      matrix[N2, N2] diag_delta;
      matrix[N1, N1] K;
      K = cov_exp_quad(x1, alpha, rho);
      for (n in 1:N1)
```

```
        K[n, n] = K[n,n] + square(sigma);
      L_K = cholesky_decompose(K);
      K_div_y1 = mdivide_left_tri_low(L_K, y1);
      K_div_y1 = mdivide_right_tri_low(K_div_y1', L_K)';
      k_x1_x2 = cov_exp_quad(x1, x2, alpha, rho);
      f2_mu = (k_x1_x2' * K_div_y1);
      v_pred = mdivide_left_tri_low(L_K, k_x1_x2);
      cov_f2 = cov_exp_quad(x2, alpha, rho) - v_pred' * v_pred;
      diag_delta = diag_matrix(rep_vector(delta, N2));

      f2 = multi_normal_rng(f2_mu, cov_f2 + diag_delta);
    }
    return f2;
  }
}
data {
  int<lower=1> N1;
  real x1[N1];
  vector[N1] y1;
  int<lower=1> N2;
  real x2[N2];
}
transformed data {
  vector[N1] mu = rep_vector(0, N1);
  real delta = 1e-9;
}
parameters {
  real<lower=0> rho;
  real<lower=0> alpha;
  real<lower=0> sigma;
}
model {
  matrix[N1, N1] L_K;
  {
    matrix[N1, N1] K = cov_exp_quad(x1, alpha, rho);
    real sq_sigma = square(sigma);

    // diagonal elements
    for (n1 in 1:N1)
```

```
      K[n1, n1] = K[n1, n1] + sq_sigma;

    L_K = cholesky_decompose(K);
  }

  rho ~ inv_gamma(5, 5);
  alpha ~ std_normal();
  sigma ~ std_normal();

  y1 ~ multi_normal_cholesky(mu, L_K);
}
generated quantities {
  vector[N2] f2;
  vector[N2] y2;

  f2 = gp_pred_rng(x2, y1, x1, alpha, rho, sigma, delta);
  for (n2 in 1:N2)
    y2[n2] = normal_rng(f2[n2], sigma);
}
```

Multiple-output Gaussian processes

Suppose we have observations $y_i \in \mathbb{R}^M$ observed at $x_i \in \mathbb{R}^K$. One can model the data like so:

$$\begin{aligned} y_i &\sim \text{multivariate normal } \left(f(x_i), \mathbf{I}_M \sigma^2\right) \\ f(x) &\sim \mathsf{GP}\big(m(x), K(x \mid \theta, \phi)\big) \\ K(x \mid \theta) &\in \mathbb{R}^{M \times M}, \quad f(x), m(x) \in \mathbb{R}^M \end{aligned}$$

where the $K(x, x' \mid \theta, \phi)_{[m,m']}$ entry defines the covariance between $f_m(x)$ and $f_{m'}(x')(x)$. This construction of Gaussian processes allows us to learn the covariance between the output dimensions of $f(x)$. If we parameterize our kernel K:

$$K(x, x' \mid \theta, \phi)_{[m,m']} = k\left(x, x' \mid \theta\right) k\left(m, m' \mid \phi\right)$$

then our finite dimensional generative model for the above is:

$$
\begin{aligned}
f &\sim \text{matrixnormal}\big(m(x), K(x \mid \alpha, \rho), C(\phi)\big) \\
y_{i,m} &\sim \text{normal}(f_{i,m}, \sigma) \\
f &\in \mathbb{R}^{N \times M}
\end{aligned}
$$

where $K(x \mid \alpha, \rho)$ is the exponentiated quadratic kernel we've used throughout this chapter, and $C(\phi)$ is a positive-definite matrix, parameterized by some vector ϕ.

The matrix normal distribution has two covariance matrices: $K(x \mid \alpha, \rho)$ to encode column covariance, and $C(\phi)$ to define row covariance. The salient features of the matrix normal are that the rows of the matrix f are distributed:

$$
f_{[n,]} \sim \text{multivariate normal}\big(m(x)_{[n,]}, K(x \mid \alpha, \rho)_{[n,n]} C(\phi)\big)
$$

and that the columns of the matrix f are distributed:

$$
f_{[,m]} \sim \text{multivariate normal}\big(m(x)_{[,m]}, K(x \mid \alpha, \rho) C(\phi)_{[m,m]}\big)
$$

This also means means that $\mathbb{E}\left[f^T f\right]$ is equal to $\text{trace}\big(K(x \mid \alpha, \rho)\big) \times C$, whereas $\mathbb{E}\left[f f^T\right]$ is $\text{trace}(C) \times K(x \mid \alpha, \rho)$. We can derive this using properties of expectation and the matrix normal density.

We should set α to 1.0 because the parameter is not identified unless we constrain $\text{trace}(C) = 1$. Otherwise, we can multiply α by a scalar d and C by $1/d$ and our likelihood will not change.

We can generate a random variable f from a matrix normal density in $\mathbb{R}^{N \times M}$ using the following algorithm:

$$
\begin{aligned}
\eta_{i,j} &\sim \text{normal}(0, 1)\, \forall i, j \\
f &= L_{K(x|1.0,\rho)}\, \eta\, L_C(\phi)^T \\
f &\sim \text{matrixnormal}\big(0, K(x \mid 1.0, \rho), C(\phi)\big) \\
\eta &\in \mathbb{R}^{N \times M} \\
L_C(\phi) &= \texttt{cholesky\_decompose}\big(C(\phi)\big) \\
L_{K(x|1.0,\rho)} &= \texttt{cholesky\_decompose}\big(K(x \mid 1.0, \rho)\big)
\end{aligned}
$$

This can be implemented in Stan using a latent-variable GP formulation. We've used LKJCorr for $C(\phi)$, but any positive-definite matrix will do.

```
data {
  int<lower=1> N;
  int<lower=1> D;
  real x[N];
  matrix[N, D] y;
}
transformed data {
  real delta = 1e-9;
}
parameters {
  real<lower=0> rho;
  vector<lower=0>[D] alpha;
  real<lower=0> sigma;
  cholesky_factor_corr[D] L_Omega;
  matrix[N, D] eta;
}
model {
  matrix[N, D] f;
  {
    matrix[N, N] K = cov_exp_quad(x, 1.0, rho);
    matrix[N, N] L_K;

    // diagonal elements
    for (n in 1:N)
      K[n, n] = K[n, n] + delta;

    L_K = cholesky_decompose(K);
    f = L_K * eta
        * diag_pre_multiply(alpha, L_Omega)';
  }

  rho ~ inv_gamma(5, 5);
  alpha ~ std_normal();
  sigma ~ std_normal();
  L_Omega ~ lkj_corr_cholesky(3);
  to_vector(eta) ~ std_normal();

  to_vector(y) ~ normal(to_vector(f), sigma);
}
```

```
generated quantities {
  matrix[D, D] Omega;
  Omega = L_Omega * L_Omega';
}
```

11. Directions, Rotations, and Hyperspheres

Directional statistics involve data and/or parameters that are constrained to be directions. The set of directions forms a sphere, the geometry of which is not smoothly mappable to that of a Euclidean space because you can move around a sphere and come back to where you started. This is why it is impossible to make a map of the globe on a flat piece of paper where all points that are close to each other on the globe are close to each other on the flat map. The fundamental problem is easy to visualize in two dimensions, because as you move around a circle, you wind up back where you started. In other words, 0 degrees and 360 degrees (equivalently, 0 and 2π radians) pick out the same point, and the distance between 359 degrees and 2 degrees is the same as the distance between 137 and 140 degrees.

Stan supports directional statistics by providing a unit-vector data type, the values of which determine points on a hypersphere (circle in two dimensions, sphere in three dimensions).

11.1. Unit Vectors

The length of a vector $x \in \mathbb{R}^K$ is given by

$$\|x\| = \sqrt{x^\top x} = \sqrt{x_1^2 + x_2^2 + \cdots + x_K^2}.$$

Unit vectors are defined to be vectors of unit length (i.e., length one).

With a variable declaration such as

```
unit_vector[K] x;
```

the value of `x` will be constrained to be a vector of size `K` with unit length; the reference manual chapter on constrained parameter transforms provides precise definitions.

Warning: An extra term gets added to the log density to ensure the distribution on unit vectors is proper. This is not a problem in practice, but it may lead to misunderstandings of the target log density output (`lp__` in some interfaces). The underlying source of the problem is that a unit vector of size K has only $K-1$ degrees

of freedom. But there is no way to map those $K-1$ degrees of freedom continuously to $\mathbb{R}^N$—for example, the circle can't be mapped continuously to a line so the limits work out, nor can a sphere be mapped to a plane. A workaround is needed instead. Stan's unit vector transform uses K unconstrained variables, then projects down to the unit hypersphere. Even though the hypersphere is compact, the result would be an improper distribution. To ensure the unit vector distribution is proper, each unconstrained variable is given a "Jacobian" adjustment equal to an independent standard normal distribution. Effectively, each dimension is drawn standard normal, then they are together projected down to the hypersphere to produce a unit vector. The result is a proper uniform distribution over the hypersphere.

11.2. Circles, Spheres, and Hyperspheres

An n-sphere, written S^n, is defined as the set of $(n+1)$-dimensional unit vectors,

$$S^n = \left\{ x \in \mathbb{R}^{n+1} \ : \ \|x\| = 1 \right\}.$$

Even though S^n is made up of points in $(n+1)$ dimensions, it is only an n-dimensional manifold. For example, S^2 is defined as a set of points in $\mathbb{R}^3$, but each such point may be described uniquely by a latitude and longitude. Geometrically, the surface defined by S^2 in $\mathbb{R}^3$ behaves locally like a plane, i.e., $\mathbb{R}^2$. However, the overall shape of S^2 is not like a plane in that is compact (i.e., there is a maximum distance between points). If you set off around the globe in a "straight line" (i.e., a geodesic), you wind up back where you started eventually; that is why the geodesics on the sphere (S^2) are called "great circles," and why we need to use some clever representations to do circular or spherical statistics.

Even though S^{n-1} behaves locally like $\mathbb{R}^{n-1}$, there is no way to smoothly map between them. For example, because latitude and longitude work on a modular basis (wrapping at 2π radians in natural units), they do not produce a smooth map.

Like a bounded interval (a, b), in geometric terms, a sphere is compact in that the distance between any two points is bounded.

11.3. Transforming to Unconstrained Parameters

Stan (inverse) transforms arbitrary points in $\mathbb{R}^{K+1}$ to points in S^K using the auxiliary variable approach of Marsaglia (1972). A point $y \in \mathbb{R}^K$ is transformed to a point $x \in S^{K-1}$ by

$$x = \frac{y}{\sqrt{y^\top y}}.$$

The problem with this mapping is that it's many to one; any point lying on a vector out of the origin is projected to the same point on the surface of the sphere. Marsaglia (1972) introduced an auxiliary variable interpretation of this mapping that provides the desired properties of uniformity; the reference manual contains the precise definitions used in the chapter on constrained parameter transforms.

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 small interval around zero, which is an option built into all of the Stan interfaces.

11.4. Unit Vectors and Rotations

Unit vectors correspond directly to angles and thus to rotations. This is easy to see in two dimensions, where a point on a circle determines a compass direction, or equivalently, an angle θ). Given an angle θ, a matrix can be defined, the pre-multiplication by which rotates a point by an angle of θ. For angle θ (in two dimensions), the 2×2 rotation matrix is defined by

$$R_\theta = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}.$$

Given a two-dimensional vector x, $R_\theta\, x$ is the rotation of x (around the origin) by θ degrees.

Unit vector type

In Stan, unit vectors in K dimensions are declared as

```
unit_vector[K] alpha;
```

A unit vector has length one (meaning the sum of squared values is one, not that its number of elements is one).

Angles from unit vectors

Angles can be calculated from unit vectors. For example, a random variable `theta` representing an angle in $(-\pi, \pi)$ radians can be declared as a two-dimensional unit vector then transformed to an angle.

```
parameters {
  unit_vector[2] xy;
}
transformed parameters {
  real<lower = -pi(), upper = pi()> theta = atan2(xy[2], xy[1]);
}
```

If the distribution of (x, y) is uniform over a circle, then the distribution of $\arctan \frac{y}{x}$ is uniform over $(-\pi, \pi]$.

It might be tempting to try to just declare theta directly as a parameter with the lower and upper bound constraint as given above. The drawback to this approach is that the values $-\pi$ and π are at $-\infty$ and ∞ on the unconstrained scale, which can produce multimodal posterior distributions when the true distribution on the circle is unimodal.

With a little additional work on the trigonometric front, the same conversion back to angles may be accomplished in more dimensions.

11.5. Circular Representations of Days and Years

A 24-hour clock naturally represents the progression of time through the day, moving from midnight to noon and back again in one rotation. A point on a circle divided into 24 hours is thus a natural representation for the time of day. Similarly, years cycle through the seasons and return to the season from which they started.

In human affairs, temporal effects often arise by convention. These can be modeled directly with ad-hoc predictors for holidays and weekends, or with data normalization back to natural scales for daylight savings time.

12. Solving Algebraic Equations

Stan provides a built-in mechanism for specifying and solving systems of algebraic equations, using the Powell hybrid method (Powell 1970). The function signatures for Stan's algebraic solver are fully described in the algebraic solver section of the reference manual.

Solving any system of algebraic equations can be translated into a root-finding problem, that is, given a function f, we wish to find y such that $f(y) = 0$.

12.1. Example: System of Nonlinear Algebraic Equations

For systems of linear algebraic equations, we recommend solving the system using matrix division. The algebraic solver becomes handy when we want to solve nonlinear equations.

As an illustrative example, we consider the following nonlinear system of two equations with two unknowns:

$$z_1 = y_1 - \theta_1$$
$$z_2 = y_1 y_2 + \theta_2$$

Our goal is to simultaneously solve all equations for y_1 and y_2, such that the vector z goes to 0.

12.2. Coding an Algebraic System

A system of algebraic equations is coded directly in Stan as a function with a strictly specified signature. For example, the nonlinear system given above can be coded using the following function in Stan (see the user-defined functions section for more information on coding user-defined functions).

```
vector system(vector y,        // unknowns
              vector theta,    // parameters
              real[] x_r,      // data (real)
              int[] x_i) {     // data (integer)
  vector[2] z;
```

```
  z[1] = y[1] - theta[1];
  z[2] = y[1] * y[2] - theta[2];
  return z;
}
```

The function takes the unknowns we wish to solve for in `y` (a vector), the system parameters in `theta` (a vector), the real data in `x_r` (a real array) and the integer data in `x_i` (an integer array). The system function returns the value of the function (a vector), for which we want to compute the roots. Our example does not use real or integer data. Nevertheless, these unused arguments must be included in the system function with exactly the signature above.

The body of the system function here could also be coded using a row vector constructor and transposition,

```
return [ y[1] - theta[1],
         y[1] * y[2] - theta[2] ]';
```

As systems get more complicated, naming the intermediate expressions goes a long way toward readability.

Strict Signature

The function defining the system must have exactly these argument types and return type. This may require passing in zero-length arrays for data or a zero-length vector for parameters if the system does not involve data or parameters.

12.3. Calling the Algebraic Solver

Let's suppose $\theta = (3, 6)$. To call the algebraic solver, we need to provide an initial guess. This varies on a case-by-case basis, but in general a good guess will speed up the solver and, in pathological cases, even determine whether the solver converges or not. If the solver does not converge, the metropolis proposal gets rejected and a warning message, stating no acceptable solution was found, is issued.

The solver has three tuning parameters to determine convergence: the relative tolerance, the function tolerance, and the maximum number of steps. Their behavior is explained in the section about algebraic solvers with control parameters.

The following code returns the solution to our nonlinear algebraic system:

```
transformed data {
  vector[2] y_guess = {1, 1};
```

```
  real x_r[0];
  int x_i[0];
}

transformed parameters {
  vector[2] theta = {3, 6};
  vector[2] y;

  y = algebra_solver(system, y_guess, theta, x_r, x_i);
}
```

which returns $y = (3, -2)$.

Data versus Parameters

The arguments for the real data `x_r` and the integer data `x_i` must be expressions that only involve data or transformed data variables. `theta`, on the other hand, must only involve parameters. Note there are no restrictions on the initial guess, `y_guess`, which may be a data or a parameter vector.

Length of the Algebraic Function and of the Vector of Unknowns

The Jacobian of the solution with respect to the parameters is computed using the implicit function theorem, which imposes certain restrictions. In particular, the Jacobian of the algebraic function f with respect to the unknowns x must be invertible. This requires the Jacobian to be square, meaning $f(y)$ and y have the same length or, in other words *the number of equations in the system is the same as the number of unknowns.*

Pathological Solutions

Certain systems may be degenerate, meaning they have multiple solutions. The algebraic solver will not report these cases, as the algorithm stops once it has found an acceptable solution. The initial guess will often determine which solution gets found first. The degeneracy may be broken by putting additional constraints on the solution. For instance, it might make "physical sense" for a solution to be positive or negative.

On the other hand, a system may not have a solution (for a given point in the parameter space). In that case, the solver will not converge to a solution. When the solver fails to do so, the current metropolis proposal gets rejected.

12.4. Control Parameters for the Algebraic Solver

The call to the algebraic solver shown above uses the default control settings. The solver allows three additional parameters, all of which must be supplied if any of them is supplied.

```
y = algebra_solver(system, y_guess, theta, x_r, x_i,
                   rel_tol, f_tol, max_steps);
```

The three control arguments are relative tolerance, function tolerance, and maximum number of steps. Both tolerances need to be satisfied. If one of them is not met, the metropolis proposal gets rejected with a warning message explaining which criterion was not satisfied. The default values for the control arguments are respectively `rel_tol = 1e-10` (10^{-10}), `f_tol = 1e-6` (10^{-6}), and `max_steps = 1e3` (10^{3}).

Tolerance

The relative and function tolerances control the accuracy of the solution generated by the solver. Relative tolerances are relative to the solution value. The function tolerance is the norm of the algebraic function, once we plug in the proposed solution. This norm should go to 0 (equivalently, all elements of the vector function are 0). It helps to think about this geometrically. Ideally the output of the algebraic function is at the origin; the norm measures deviations from this ideal. As the length of the return vector increases, a certain function tolerance becomes an increasingly difficult criterion to meet, given each individual element of the vector contribute to the norm.

Smaller relative tolerances produce more accurate solutions but require more computational time.

Sensitivity Analysis

The tolerances should be set low enough that setting them lower does not change the statistical properties of posterior samples generated by the Stan program.

Maximum Number of Steps

The maximum number of steps can be used to stop a runaway simulation. This can arise in MCMC when a bad jump is taken, particularly during warmup. If the limit is hit, the current metropolis proposal gets rejected. Users will see a warning message stating the maximum number of steps has been exceeded.

13. Ordinary Differential Equations

Stan provides a built-in mechanism for specifying and solving systems of ordinary differential equations (ODEs). Stan provides two different integrators, one tuned for solving non-stiff systems and one for stiff systems.

- `rk45`: a fourth and fifth order Runge-Kutta method for non-stiff systems (Dormand and Prince 1980; Ahnert and Mulansky 2011), and
- `bdf`: a variable-step, variable-order, backward-differentiation formula implementation for stiff systems (Cohen and Hindmarsh 1996; Serban and Hindmarsh 2005)

For a discussion of stiff ODE systems, see the stiff ODE section. In a nutshell, the stiff solvers are slower, but more robust; how much so depends on the system and the region of parameter space. The function signatures for Stan's ODE solvers can be found in the reference manual section on ODE solvers.

13.1. Example: Simple Harmonic Oscillator

As an example of a system of ODEs, consider a harmonic oscillator, which is characterized by an equilibrium position and a restoring force proportional to the displacement with friction. The system state will be a pair $y = (y_1, y_2)$ representing position and momentum: a point in phase space. The change in the system with respect to time is given by the following differential equations.[1]

$$\frac{d}{dt}y_1 = y_2 \qquad \frac{d}{dt}y_2 = -y_1 - \theta y_2$$

The state equations implicitly define the system state at a given time as a function of an initial state, elapsed time since the initial state, and the system parameters.

Solutions Given Initial Conditions

Given a value of the system parameter θ and an initial state $y(t_0)$ at time t_0, it is possible to simulate the evolution of the solution numerically in order to calculate $y(t)$ for a specified sequence of times $t_0 < t_1 < t_2 < \cdots$.

[1] This example is drawn from the documentation for the Boost Numeric Odeint library (Ahnert and Mulansky 2011), which Stan uses to implement the `rk45` solver.

13.2. Coding an ODE System

A system of ODEs is coded directly in Stan as a function with a strictly specified signature. For example, the simple harmonic oscillator can be coded using the following function in Stan (see the user-defined functions chapter for more information on coding user-defined functions).

```
real[] sho(real t,        // time
           real[] y,      // state
           real[] theta,  // parameters
           real[] x_r,    // data (real)
           int[] x_i) {   // data (integer)
  real dydt[2];
  dydt[1] = y[2];
  dydt[2] = -y[1] - theta[1] * y[2];
  return dydt;
}
```

The function takes in a time `t` (a real value), a a system state `y` (real array), system parameters `theta` (a real array), along with real data in variable `x_r` (a real array) and integer data in variable `x_i` (an integer array). The system function returns the array of derivatives of the system state with respect to time, evaluated at time `t` and state `y`. The simple harmonic oscillator coded here does not have time-sensitive equations; that is, `t` does not show up in the definition of `dydt`. The simple harmonic oscillator does not use real or integer data, either. Nevertheless, these unused arguments must be included as arguments in the system function with exactly the signature shown above.

Strict Signature

The function defining the system must have exactly these argument types and return type. This may require passing in zero-length arrays for data or parameters if the system does not involve data or parameters. A full example for the simple harmonic oscillator, which does not depend on any constant data variables, is provided in the simple harmonic oscillator trajectory plot.

Discontinuous ODE System Function

The ODE integrator is able to integrate over discontinuities in the state function, although the accuracy of points near the discontinuity may be problematic (requiring many small steps). An example of such a discontinuity is a lag in a pharmacokinetic model, where a concentration is going to be zero for times $0 < t < t'$ for some

lag-time t', whereas it will be nonzero for times $t \geq t'$. In this example example, we would use code in the system such as

```
if (t < t_lag)
  return 0;
else
  ... return non-zero value...;
```

13.3. Solving a System of Linear ODEs using a Matrix Exponential

The solution to $\frac{d}{dt}y = ay$ is $y = y_0 e^{at}$, where the constant y_0 is determined by boundary conditions. We can extend this solution to the vector case:

$$\frac{d}{dt}y = A\,y$$

where y is now a vector of length n and A is an n by n matrix. The solution is then given by:

$$y = e^{tA}\,y_0$$

where the matrix exponential is formally defined by the convergent power series:

$$e^{tA} = \sum_{n=0}^{\infty} \frac{tA^n}{n!} = I + tA + \frac{t^2A^2}{2!} + \cdots$$

We can apply this technique to the simple harmonic oscillator example, by setting

$$y = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} \qquad A = \begin{bmatrix} 0 & 1 \\ -1 & -\theta \end{bmatrix}$$

The Stan model to simulate noisy observations using a matrix exponential function is given below. Because we are performing matrix operations, we declare `y0` and `y_hat` as vectors, instead of using arrays, as in the previous example code.

In general, computing a matrix exponential will be more efficient than using a numerical solver. We can however only apply this technique to systems of linear ODEs.

```
data {
  int<lower=1> T;
  vector[2] y0;
```

```
  real ts[T];
  real theta[1];
}
model {
}
generated quantities {
  vector[2] y_hat[T];
  matrix[2, 2] A = [[ 0,  1],
                    [-1, -theta[1]]]
  for (t in 1:T)
    y_hat[t] = matrix_exp((t - 1) * A) * y0;
  // add measurement error
  for (t in 1:T) {
    y_hat[t, 1] += normal_rng(0, 0.1);
    y_hat[t, 2] += normal_rng(0, 0.1);
  }
}
```

This Stan program simulates noisy measurements from a simple harmonic oscillator. The system of linear differential equations is coded as a matrix. The system parameters `theta` and initial state `y0` are read in as data along observation times `ts`. The generated quantities block is used to solve the ODE for the specified times and then add random measurement error, producing observations `y_hat`. Because the ODEs are linear, we can use the `matrix_exp` function to solve the system.

13.4. Measurement Error Models

Statistical models or differential equations may be used to estimate the parameters and/or initial state of a dynamic system given noisy measurements of the system state at a finite number of time points.

For instance, suppose the simple harmonic oscillator has a parameter value of $\theta = 0.15$ and initial state $y(t = 0) = (1, 0)$. Now suppose the system is observed at 10 time points, say $t = 1, 2, ..., 10$, where each measurement of $y(t)$ has independent $\mathsf{normal}(0, 0.1)$ error in both dimensions ($y_1(t)$ and $y_2(t)$). A plot of such measurements is shown in the simple harmonic oscillator trajectory plots.

Trajectory of the simple harmonic oscillator given parameter $\theta = 0.15$ and initial condition $y(t = 0) = (1, 0)$ with additional independent $\mathsf{normal}(0, 0.1)$ measurement error in both dimensions.

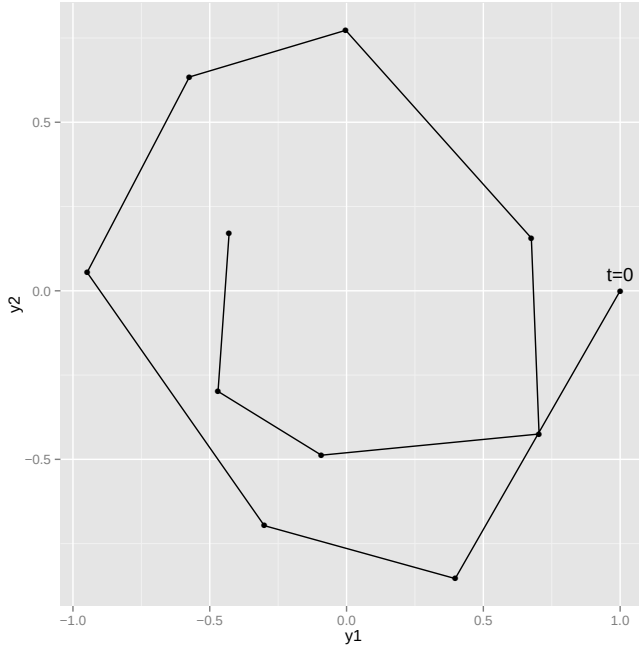

Figure 13.1: Simple harmonic oscillator trajectory

Simulating Noisy Measurements

The data used to make this plot is derived from the Stan model to simulate noisy observations given below.

```
functions {
  real[] sho(real t,
             real[] y,
             real[] theta,
             real[] x_r,
             int[] x_i) {
    real dydt[2];
    dydt[1] = y[2];
    dydt[2] = -y[1] - theta[1] * y[2];
    return dydt;
  }
```

```
}
data {
  int<lower=1> T;
  real y0[2];
  real t0;
  real ts[T];
  real theta[1];
}
transformed data {
  real x_r[0];
  int x_i[0];
}
model {
}
generated quantities {
  real y_hat[T,2] = integrate_ode_rk45(sho, y0, t0, ts, theta, x_r, x_i);
  // add measurement error
  for (t in 1:T) {
    y_hat[t, 1] += normal_rng(0, 0.1);
    y_hat[t, 2] += normal_rng(0, 0.1);
  }
}
```

The system of differential equations is coded as a function. The system parameters `theta` and initial state `y0` are read in as data along with the initial time `t0` and observation times `ts`. The generated quantities block is used to solve the ODE for the specified times and then add random measurement error, producing observations `y_hat`. Because the system is not stiff, the `rk45` solver is used. Note that when the system is linear, it is a good idea to try using the matrix exponential function, as the method can, depending on the system at hand, yield more efficient results.

This program illustrates the way in which the ODE solver is called in a Stan program,

```
y_hat = integrate_ode_rk45(sho, y0, t0, ts, theta, x_r, x_i);
```

This assigns the solutions to the system defined by function `sho`, given initial state `y0`, initial time `t0`, requested solution times `ts`, parameters `theta`, real data `x`, and integer data `x_int`. The call explicitly specifies the Runge-Kutta solver (for non-stiff systems).

Here, the ODE solver is called in the generated quantities block to provide a 10×2 array of solutions `y_hat` to which measurement error is added using the normal

pseudo-random number generating function `normal_rng`. The number of rows in the solution array is the same as the size of `ts`, the requested solution times.

Data versus Parameters

Unlike other functions, the integration functions for ODEs are limited as to the origins of variables in their arguments. In particular, real data `x`, and integer data `x_int` must be expressions that only involve data or transformed data variables. The initial state `y`, the initial time `t0`, the requested solution times `ts`, or the parameters `theta` may involve parameters.

Estimating System Parameters and Initial State

Stan provides statistical inference for unknown initial states and/or parameters. The ODE solver will be used deterministically to produce predictions, much like the linear predictor does in a generalized linear model. These states will then be observed with measurement error.

```
functions {
  real[] sho(real t,
             real[] y,
             real[] theta,
             real[] x_r,
             int[] x_i) {
    real dydt[2];
    dydt[1] = y[2];
    dydt[2] = -y[1] - theta[1] * y[2];
    return dydt;
  }
}
data {
  int<lower=1> T;
  real y[T,2];
  real t0;
  real ts[T];
}
transformed data {
  real x_r[0];
  int x_i[0];
}
parameters {
```

```
  real y0[2];
  vector<lower=0>[2] sigma;
  real theta[1];
}
model {
  real y_hat[T,2];
  sigma ~ cauchy(0, 2.5);
  theta ~ std_normal();
  y0 ~ std_normal();
  y_hat = integrate_ode_rk45(sho, y0, t0, ts, theta, x_r, x_i);
  for (t in 1:T)
    y[t] ~ normal(y_hat[t], sigma);
}
```

This Stan program allows estimates of unknown initial conditions `y0` and system parameter `theta` for the simple harmonic oscillator with independent normal measurement error.

A Stan program that can be used to estimate both the initial state and parameter value for the simple harmonic oscillator given noisy observations is given above. Compared to the program for simulation, the program to estimate parameters uses the `integrate_ode` function in the model block rather than the generated quantities block. There are Cauchy priors on the measurement error scales `sigma` and standard normal priors on the components of parameter array `theta` and initial state parameter array `y0`. The solutions to the ODE are then assigned to an array `y_hat`, which is then used as the location in the observation noise model as follows.

```
y_hat = integrate_ode_rk45(sho, y0, t0, ts, theta, x_r, x_i);
for (t in 1:T)
  y[t] ~ normal(y_hat[t], sigma);
```

As with other regression-like models, it's easy to change the noise model to be robust (e.g., Student-t distributed), to be correlated in the state variables (e.g., with a multivariate normal distribution), or both (e.g., with a multivariate Student-t distribution).

In this simple model with independent noise scales of 0.10, 10 observed data points for times $t = 1, \ldots, 10$ is sufficient to reliably estimate the ODE parameter, initial state, and noise scales.

13.5. Stiff ODEs

A stiff system of ordinary differential equations can be roughly characterized as systems presenting numerical difficulties for gradient-based stepwise solvers. Stiffness typically arises due to varying curvature in the dimensions of the state, for instance one component evolving orders of magnitude more slowly than another.[2]

Stan provides a specialized solver for stiff ODEs (Cohen and Hindmarsh 1996; Serban and Hindmarsh 2005). An ODE system is specified exactly the same way with a function of exactly the same signature. The only difference is in the call to the integrator for the solution; the `rk45` suffix is replaced with `bdf`, as in

```
y_hat = integrate_ode_bdf(sho, y0, t0, ts, theta, x_r, x_i);
```

Using the stiff (`bdf`) integrator on a system that is not stiff may be much slower than using the non-stiff (`rk45`) integrator; this is because it computes additional Jacobians to guide the integrator. On the other hand, attempting to use the non-stiff integrator for a stiff system will fail due to requiring a small step size and too many steps.

13.6. Control Parameters for ODE Solving

The calls to the integrators shown above just used the default control settings. Both the non-stiff and stiff integrators allow three additional arguments, all of which must be supplied if any of them is required.

```
y_hat = integrate_ode_bdf(sho, y0, t0, ts, theta, x_r, x_i,
                          rel_tol, abs_tol, max_steps);
```

The three control arguments are relative tolerance, absolute tolerance, and maximum number of steps. The default values for relative and absolute tolerance are both `1e-6` (10^{-6}), and the default maximum number of steps is `1e6` (10^6).

Data only for control parameters

The control parameters must be data variables—they can not be parameters or expressions that depend on parameters, including local variables in any block other than transformed data and generated quantities. User-defined function arguments may be qualified as only allowing data arguments using the `data` qualifier.

[2] Not coincidentally, high curvature in the posterior of a general Stan model poses the same kind of problem for Euclidean Hamiltonian Monte Carlo (HMC) sampling. The reason is that HMC is based on the leapfrog algorithm, a gradient-based, stepwise numerical differential equation solver specialized for Hamiltonian systems with separable potential and kinetic energy terms.

Tolerance

The relative and absolute tolerance control the accuracy of the solutions generated by the integrator. Relative tolerances are relative to the solution value, whereas absolute tolerances is the maximum absolute error allowed in a solution.

Smaller tolerances produce more accurate solutions. Smaller tolerances also require more computation time.

Sensitivity Analysis

The tolerances should be set low enough that setting them lower does not change the statistical properties of posterior samples generated by the Stan program.

Maximum Number of Steps

The maximum number of steps can be used to stop a runaway simulation. This can arise in MCMC when a bad jump is taken, particularly during warmup. With the non-stiff solver, this may result in jumping into a stiff region of the parameter space, which would require a small step size and many steps to satisfy even modest tolerances.

14. Computing One Dimensional Integrals

Definite and indefinite one dimensional integrals can be performed in Stan using the `integrate_1d` function.

As an example, the normalizing constant of a left-truncated normal distribution is

$$\int_a^\infty \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2}\frac{(x-\mu)^2}{\sigma^2}}$$

To compute this integral in Stan, the integrand must first be defined as a Stan function (see the Stan Reference Manual chapter on User-Defined Functions for more information on coding user-defined functions).

```
real normal_density(real x,          // Function argument
                    real xc,         // Complement of function argument
                                     //  on the domain (defined later)
                    real[] theta,    // parameters
                    real[] x_r,      // data (real)
                    int[] x_i) {     // data (integer)
  real mu = theta[1];
  real sigma = theta[2];

  return 1 / (sqrt(2 * pi()) * sigma) * exp(-0.5 * ((x - mu) / sigma)^2);
}
```

This function is expected to return the value of the integrand evaluated at point `x`. The argument `xc` is used in definite integrals to avoid loss of precision near the limits of integration and is set to NaN when either limit is infinite (see the section on precision/loss in the chapter on Higher-Order Functions of the Stan Functions Reference for details on how to use this). The argument `theta` is used to pass in arguments of the integral that are a function of the parameters in our model. The arguments `x_r` and `x_i` are used to pass in real and integer arguments of the integral that are not a function of our parameters.

The function defining the integrand must have exactly the argument types and return type of `normal_density` above, though argument naming is not important. Even if `x_r` and `x_i` are unused in the integrand, they must be included in the function

signature. This may require passing in zero-length arrays for data or a zero-length vector for parameters if the integral does not involve data or parameters.

14.1. Calling the Integrator

Suppose that our model requires evaluating the lpdf of a left-truncated normal, but the truncation limit is to be estimated as a parameter. Because the truncation point is a parameter, we must include the normalization term of the truncated pdf when computing our model's log density. Note this is just an example of how to use the 1D integrator. The more efficient way to perform the correct normalization in Stan is described in the chapter on Truncated or Censored Data of this guide.

Such a model might look like (include the function defined at the beginning of this chapter to make this code compile):

```
data {
  int N;
  real y[N];
}

transformed data {
  real x_r[0];
  int x_i[0];
}

parameters {
  real mu;
  real<lower = 0.0> sigma;
  real left_limit;
}

model {
  mu ~ normal(0, 1);
  sigma ~ normal(0, 1);
  left_limit ~ normal(0, 1);
  target += normal_lpdf(y | mu, sigma);
  target += log(integrate_1d(normal_density,
                             left_limit,
                             positive_infinity(),
                             { mu, sigma }, x_r, x_i));
}
```

Limits of Integration

The limits of integration can be finite or infinite. The infinite limits are made available via the Stan calls `negative_infinity()` and `positive_infinity()`.

If both limits are either `negative_infinity()` or `positive_infinity()`, the integral and its gradients are set to zero.

Data Versus Parameters

The arguments for the real data `x_r` and the integer data `x_i` must be expressions that only involve data or transformed data variables. `theta`, on the other hand, can be a function of data, transformed data, parameters, or transformed parameters.

The endpoints of integration can be data or parameters (and internally the derivatives of the integral with respect to the endpoints are handled with the Leibniz integral rule).

14.2. Integrator Convergence

The integral is performed with the iterative 1D quadrature methods implemented in the Boost library (Agrawal et al. 2017). If the nth estimate of the integral is denoted I_n and the nth estimate of the norm of the integral is denoted $|I|_n$, the iteration is terminated when

$$\frac{|I_{n+1} - I_n|}{|I|_{n+1}} < \text{relative tolerance}$$

The `relative_tolerance` parameter can be optionally specified as the last argument to `integrate_1d`. By default, `integrate_1d` follows the Boost library recommendation of setting `relative_tolerance` to the square root of the machine epsilon of double precision floating point numbers (about `1e-8`).

Zero-crossing Integrals

Integrals on the (possibly infinite) interval (a, b) that cross zero are split into two integrals, one from $(a, 0)$ and one from $(0, b)$. This is because the quadrature methods employed internally can have difficulty near zero.

In this case, each integral is separately integrated to the given `relative_tolerance`.

Avoiding precision loss near limits of integration in definite integrals

If care is not taken, the quadrature can suffer from numerical loss of precision near the endpoints of definite integrals.

For instance, in integrating the pdf of a beta distribution when the values of α and β are small, most of the probability mass is lumped near zero and one.

The pdf of a beta distribution is proportional to

$$p(x) \propto x^{\alpha-1}(1-x)^{\beta-1}$$

Normalizing this distribution requires computing the integral of $p(x)$ from zero to one. In Stan code, the integrand might look like:

```
real beta(real x, real xc, real[] theta, real[] x_r, int[] x_i) {
  real alpha = theta[1];
  real beta = theta[2];

  return x^(alpha - 1.0) * (1.0 - x)^(beta - 1.0);
}
```

The issue is that there will be numerical breakdown in the precision of `1.0 - x` as `x` gets close to one. This is because of the limited precision of double precision floating numbers. This integral will fail to converge for values of `alpha` and `beta` much less than one.

This is where `xc` is useful. It is defined, for definite integrals, as a high precision version of the distance from `x` to the nearest endpoint. To make use of this for the beta integral, the integrand can be re-coded:

```
real beta(real x, real xc, real[] theta, real[] x_r, int[] x_i) {
  real alpha = theta[1];
  real beta = theta[2];
  real v;

  if(x > 0.5) {
    v = x^(alpha - 1.0) * xc^(beta - 1.0);
  } else {
    v = x^(alpha - 1.0) * (1.0 - x)^(beta - 1.0);
  }
```

```
  return v;
}
```

This version of the integrand will converge for much smaller values of `alpha` and `beta` than otherwise possible.

Note, `xc` is only used for definite integrals. If either the left endpoint is at negative infinity or the right endpoint is at positive infinity, `xc` will be NaN.

For zero-crossing definite integrals (see section Zero Crossing) the integrals are broken into two pieces ($(a, 0)$ and $(0, b)$ for endpoints $a < 0$ and $b > 0$) and `xc` is a high precision version of the distance to the limits of each of the two integrals separately. This means `xc` will be the a high precision version of `a - x`, `x`, or `b - x`, depending on the value of `x` and the endpoints.

Part 2. Programming Techniques

This part of the manual surveys general programming techniques in Stan that are useful across a range of different model types.

15. Floating Point Arithmetic

Computers approximate real values in $\mathbb{R}$ using a fixed number of bits. This chapter explains how this is done and why it is important for writing robust Stan (and other numerical) programs. The subfield of computer science devoted to studying how real arithmetic works on computers is called *numerical analysis*.

15.1. Floating-point representations

Stan's arithmetic is implemented using double-precision arithmetic. The behavior of most[1] modern computers follows the floating-point arithmetic, *IEEE Standard for Floating-Point Arithmetic* (IEEE 754).

Finite values

The double-precision component of the IEEE 754 standard specifies the representation of real values using a fixed pattern of 64 bits (8 bytes). All values are represented in base two (i.e., binary). The representation is divided into two signed components:

- *significand* (53 bits): base value representing significant digits
- *exponent* (11 bits): power of two multiplied by the base

The *value* of a finite floating point number is

$$v = (-1)^s \times c\, 2^q$$

Normality

A *normal* floating-point value does not use any leading zeros in its significand; *subnormal* numbers may use leading zeros. Not all I/O systems support subnormal numbers.

Ranges and extreme values

There are some reserved exponent values so that legal exponent values range between $-(2^{10}) + 2 = -1022$ and $2^{10} - 1 = 1023$. Legal significand values are

[1] The notable exception is Intel's optimizing compilers under certain optimization settings.

between -2^{52} and $2^{52} - 1$. Floating point allows the representation of both really big and really small values. Some extreme values are

- *largest normal finite number*: $\approx 1.8 \times 10^{308}$
- *largest subnormal finite number*: $\approx 2.2 \times 10^{308}$
- *smallest positive normal number*: $\approx 2.2 \times 10^{-308}$
- *smallest positive subnormal number*: $\approx 4.9 \times 10^{-324}$

Signed zero

Because of the sign bit, there are two ways to represent zero, often called "positive zero" and "negative zero". This distinction is irrelevant in Stan (as it is in R), because the two values are equal (i.e., `0 == -0` evaluates to true).

Not-a-number values

A specially chosen bit pattern is used for the *not-a-number* value (often written as `NaN` in programming language output, including Stan's).

Stan provides a value function `not_a_number()` that returns this special not-a-number value. It is meant to represent error conditions, not missing values. Usually when not-a-number is an argument to a function, the result will not-a-number if an exception (a rejection in Stan) is not raised.

Stan also provides a test function `is_nan(x)` that returns 1 if `x` is not-a-number and 0 otherwise.

Not-a-number values propagate under almost all mathematical operations. For example, all of the built-in binary arithmetic operations (addition, subtraction, multiplication, division, negation) return not-a-number if any of their arguments are not-a-number. The built-in functions such as `log` and `exp` have the same behavior, propagating not-a-number values.

Most of Stan's built-in functions will throw exceptions (i.e., reject) when any of their arguments is not-a-number.

Comparisons with not-a-number always return false, up to and including comparison with itself. That is, `not_a_number() == not_a_number()` somewhat confusingly returns false. That is why there is a built-in `is_nan()` function in Stan (and in C++). The only exception is negation, which remains coherent. This means `not_a_number() != not_a_number()` returns true.

Undefined operations often return not-a-number values. For example, `sqrt(-1)` will evaluate to not-a-number.

Positive and negative infinity

There are also two special values representing positive infinity (∞) and negative infinity ($-\infty$). These are not as pathological as not-a-number, but are often used to represent error conditions such as overflow and underflow. For example, rather than raising an error or returning not-a-number, `log(0)` evaluates to negative infinity. Exponentiating negative infinity leads back to zero, so that `0 == exp(log(0))`. Nevertheless, this should not be done in Stan because the chain rule used to calculate the derivatives will attempt illegal operations and return not-a-number.

There are value functions `positive_infinity()` and `negative_infinity()` as well as a test function `is_inf()`.

Positive and negative infinity have the expected comparison behavior, so that `negative_infinty() < 0` evaluates to true (represented with 1 in Stan). Also, negating positive infinity leads to negative infinity and vice-versa.

Positive infinity added to either itself or a finite value produces positive infinity. Negative infinity behaves the same way. However, attempts to subtract positive infinity from itself produce not-a-number, not zero. Similarly, attempts to divide infinite values results in a not-a-number value.

15.2. Literals: decimal and scientific notation

In programming languages such as Stan, numbers may be represented in standard *decimal* (base 10) notation. For example, `2.39` or `-1567846.276452`. Remember there is no point in writing more than 16 significant digits as they cannot be represented. A number may be coded in Stan using *scientific notation*, which consists of a signed decimal representation of a base and a signed integer decimal exponent. For example, `36.29e-3` represents the number 36.29×10^{-3}, which is the same number as is represented by `0.03629`.

15.3. Arithmetic Precision

The choice of significand provides $\log_{10} 2^{53} \approx 15.95$ decimal (base 10) digits of *arithmetic precision*. This is just the precision of the floating-point representation. After several operations are chained together, the realized arithmetic precision is often much lower.

Rounding and probabilities

In practice, the finite amount of arithmetic precision leads to *rounding*, whereby a number is represented by the closest floating-point number. For example, with only 16 decimal digits of accuracy,

```
1 + 1e-20 == 1
```

The closest floating point number to $1 + 10^{-20}$ turns out to be 1 itself. By contrast,

```
0 + 1e-20 == 1e-20
```

This highlights the fact that precision depends on scale. Even though `1 + 1e-20 == 1`, we have `1e-20 + 1e-20 == 2e-20`, as expected.

Rounding also manifests itself in a lack of *transitivity*. In particular, it does *not* usually hold for three floating point numbers a, b, c that $(a + b) + c = a + (b = c)$.

In statistical applications, problems often manifest in situations where users expect the usual rules of real-valued arithmetic to hold. Suppose we have a lower triangular matrix L with strictly positive diagonal, so that it is the Cholesky factor of a positive-definite matrix $L\,L^\top$. In practice, rounding and loss of precision may render the result $L\,L^\top$ neither symmetric nor positive definite.

In practice, care must be taken to defend against rounding. For example, symmetry may be produced by adding $L\,L^\top$ with its transpose and dividing by two, or by copying the lower triangular portion into the upper portion. Positive definiteness may be maintained by adding a small quantity to the diagonal.

Machine precision and the asymmetry of 0 and 1

The smallest number greater than zero is roughly $0 + 10^{-323}$. The largest number less than zero is roughly $1 - 10^{-15.95}$. The asymmetry is apparent when considering the representation of that largest number smaller than one—the exponent is of no help, and the number is represented as the binary equivalent of 0.9999999999999999.

For this reason, the *machine precision* is said to be roughly $10^{15.95}$. This constant is available as `machine_precision()` in Stan.

Complementary and epsilon functions

Special operations are available to mitigate this problem with numbers rounding when they get close to one. For example, consider the operation `log(1 + x)` for positive `x`. When `x` is small (less than 10^{-16} for double-precision floating point), the sum in the argument will round to 1 and the result will round to zero. To

allow more granularity, programming languages provide a library function directly implementing $f(x) = \log(1 + x)$. In Stan (as in C++), this operation is written as `log1p(x)`. Because `x` itself may be close to zero, the function `log1p(x)` can take the logarithm of values very close to one, the results of which are close to zero.

Similarly, the complementary cumulative distribution functions (CCDF), defined by $F_Y^{\complement}(y) = 1 - F_Y(y)$, where F_Y is the cumulative distribution function (CDF) for the random variable Y. This allows values very close to one to be represented in complementary form.

Catastrophic cancellation

Another downside to floating point representations is that subtraction of two numbers close to each other results in a loss of precision that depends on how close they are. This is easy to see in practice. Consider

$$\begin{aligned} & 1.23456789012345 \\ - & 1.23456789012344 \\ = & \ 0.00000000000001 \end{aligned}$$

We start with fifteen decimal places of accuracy in the arguments and are left with a single decimal place of accuracy in the result.

Catastrophic cancellation arises in statistical computations whenever we calculate variance for a distribution with small standard deviations relative to its location. When calculating summary statistics, Stan uses *Welford's algorithm* for computing variances. This avoids catastrophic cancellation and may also be carried out in a single pass.

Overflow

Even though `1e200` may be represented as a double precision floating point value, there is no finite value large enough to represent `1e200 * 1e200`. The result of `1e200 * 1e200` is said to *overflow*. The IEEE 754 standard requires the result to be positive infinity.

Overflow is rarely a problem in statistical computations. If it is, it's possible to work on the log scale, just as for underflow as described below.

Underflow and the log scale

When there is no number small enough to represent a result, it is said to *underflow*. For instance, `1e-200` may be represented, but `1e-200 * 1e-200` underflows so that the result is zero.

Underflow is a ubiquitous problem in likelihood calculations, For example, if $p(y_n \mid \theta) < 0.1$, then

$$p(y \mid \theta) = \prod_{n=1}^{N} p(y_n \mid \theta)$$

will underflow as soon as $N > 350$ or so.

To deal with underflow, work on the log scale. Even though $p(y \mid \theta)$ can't be represented, there is no problem representing

$$\begin{aligned} \log p(y \mid \theta) &= \log \textstyle\prod_{n=1}^{N} p(y_n \mid \theta) \\ &= \textstyle\sum_{n=1}^{N} \log p(y_n \mid \theta) \end{aligned}$$

This is why all of Stan's probability functions operate on the log scale.

Adding on the log scale

If we work on the log scale, multiplication is converted to addition,

$$\log(a \times b) = \log a + \log b.$$

Thus we can just start on the log scale and stay there through multiplication. But what about addition? If we have $\log a$ and $\log b$, how do we get $\log(a + b)$? Working out the algebra,

$$\log(a + b) = \log(\exp(\log a) + \exp(\log b))$$

The nested log of sum of exponentials is so common, it has its own name,

$$\texttt{log\_sum\_exp}(u, v) = \log(\exp(u) + \exp(v)).$$

so that

$$\log(a + b) = \texttt{log\_sum\_exp}(\log a, \log b).$$

Although it appears this might overflow as soon as exponentiation is introduced, evaluation does not proceed by evaluating the terms as written. Instead, with a little algebra, the terms are rearranged into a stable form,

$$\texttt{log\_sum\_exp}(u, v) = \max(u, v) + \log\big(\exp(u - \max(u, v)) + \exp(v - \max(u, v))\big).$$

Because the terms inside the exponentiations are $u - \max(u, v)$ and $v - \max(u, v)$, one will be zero, and the other will be negative. Because the operation is symmetric, it may be assumed without loss of generality that $u \geq v$, so that

$$\texttt{log\_sum\_exp}(u, v) = u + \log\big(1 + \exp(v - u)\big).$$

The inner term may itself be evaluated using `log1p`, there is only limited gain because $\exp(v - u)$ is only near zero when u is much larger than v, meaning the result is likely to round to u anyway.

To conclude, when evaluating $\log(a + b)$ given $\log a$ and $\log b$, and assuming $\log a > \log b$, return

$$\log(a + b) = \log a + \texttt{log1p}\big(\exp(\log b - \log a)\big).$$

15.4. Comparing floating-point numbers

Because floating-point representations are inexact, it is rarely a good idea to test exact inequality. The general recommendation is that rather than testing `x == y`, an approximate test may be used given an absolute or relative tolerance.

Given a positive *absolute tolerance* of `epsilon`, `x` can be compared to `y` using the conditional

```
abs(x - y) <= epsilon.
```

Absolute tolerances work when the scale of `x` and `y` and the relevant comparison is known.

Given a positive *relative tolerance* of `epsilon`, a typical comparison is

```
2 * abs(x - y) / (abs(x) + abs(y)) <= epsilon.
```

16. Matrices, Vectors, and Arrays

This chapter provides pointers as to how to choose among the various matrix, vector, and array data structures provided by Stan.

16.1. Basic Motivation

Stan provides two basic scalar types, `int` and `real`, and three basic linear algebra types, `vector`, `row_vector`, and `matrix`. Then Stan allows arrays to be of any dimension and contain any type of element (though that type must be declared and must be the same for all elements).

This leaves us in the awkward situation of having three one-dimensional containers, as exemplified by the following declarations.

```
real a[N];
vector[N] a;
row_vector[N] a;
```

These distinctions matter. Matrix types, like vector and row vector, are required for linear algebra operations. There is no automatic promotion of arrays to vectors because the target, row vector or column vector, is ambiguous. Similarly, row vectors are separated from column vectors because multiplying a row vector by a column vector produces a scalar, whereas multiplying in the opposite order produces a matrix.

The following code fragment shows all four ways to declare a two-dimensional container of size $M \times N$.

```
real b[M, N];          // b[m] : real[]     (efficient)
vector[N] b[M];        // b[m] : vector     (efficient)
row_vector[N] b[M];    // b[m] : row_vector (efficient)
matrix[M, N] b;        // b[m] : row_vector (inefficient)
```

The main differences among these choices involve efficiency for various purposes and the type of `b[m]`, which is shown in comments to the right of the declarations. Thus the only way to efficiently iterate over row vectors is to use the third declaration, but if you need linear algebra on matrices, but the only way to use matrix operations is to use the fourth declaration.

The inefficiencies due to any manual reshaping of containers is usually slight compared to what else is going on in a Stan program (typically a lot of gradient calculations).

16.2. Fixed Sizes and Indexing out of Bounds

Stan's matrices, vectors, and array variables are sized when they are declared and may not be dynamically resized. Function arguments do not have sizes, but these sizes are fixed when the function is called and the container is instantiated. Also, declarations may be inside loops and thus may change over the course of running a program, but each time a declaration is visited, it declares a fixed size object.

When an index is provided that is out of bounds, Stan throws a rejection error and computation on the current log density and gradient evaluation is halted and the algorithm is left to clean up the error. All of Stan's containers check the sizes of all indexes.

16.3. Data Type and Indexing Efficiency

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

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

Matrices vs. Two-Dimensional Arrays

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

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

Second, matrices store their data in column-major order. Furthermore, all of the data in a matrix is guaranteed to be contiguous in memory. This is an important consideration for optimized code because bringing in data from memory to cache is much more expensive than performing arithmetic operations with contemporary CPUs. Arrays, on the other hand, only guarantee that the values of primitive types

are contiguous in memory; otherwise, they hold copies of their values (which are returned by reference wherever possible).

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

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

Arrays, on the other hand, should be traversed in row-major order (i.e., last index fastest), as in the following example.

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

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

This is true even for arrays of matrices. For example, the ideal order in which to traverse a two-dimensional array of matrices is

```
matrix[M, N] b[I, J];
// ...
for (i in 1:I)
  for (j in 1:J)
    for (n in 1:N)
      for (m in 1:M)
        ... do something with b[i, j, m, n] ...
```

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

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

is much more efficient than the pure matrix version

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

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

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

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

than it is to do this

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

(Row) Vectors vs. One-Dimensional Arrays

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

16.4. Memory Locality

The key to understanding efficiency of matrix and vector representations is memory locality and reference passing versus copying.

Memory Locality

CPUs on computers bring in memory in blocks through layers of caches. Fetching from memory is *much* slower than performing arithmetic operations. The only way to make container operations fast is to respect memory locality and access elements that are close together in memory sequentially in the program.

Matrices

Matrices are stored internally in column-major order. That is, an $M \times N$ matrix stores its elements in the order

$$(1,1), (2,1), \ldots, (M,1), (1,2), \ldots, (M,2), \ldots, (1,N), \ldots, (M,N).$$

This means that it's much more efficient to write loops over matrices column by column, as in the following example.

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

It also follows that pulling a row out of a matrix is not memory local, as it has to stride over the whole sequence of values. It also requires a copy operation into a new data structure as it is not stored internally as a unit in a matrix. For sequential access to row vectors in a matrix, it is much better to use an array of row vectors, as in the following example.

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

Even if what is done involves a function call, the row vector `a[m]` will not have to be copied.

Arrays

Arrays are stored internally following their data structure. That means a two dimensional array is stored in row-major order. Thus it is efficient to pull out a "row" of a two-dimensional array.

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

A difference with matrices is that the entries `a[m]` in the two dimensional array are not necessarily adjacent in memory, so there are no guarantees on iterating over all the elements in a two-dimensional array will provide memory locality across the "rows."

16.5. Converting among Matrix, Vector, and Array Types

There is no automatic conversion among matrices, vectors, and arrays in Stan. But there are a wide range of conversion functions to convert a matrix into a vector, or a multi-dimensional array into a one-dimensional array, or convert a vector to an array. See the section on mixed matrix and array operations in the functions reference manual for a complete list of conversion operators and the multi-indexing chapter for some reshaping operations involving multiple indexing and range indexing.

16.6. Aliasing in Stan Containers

Stan expressions are all evaluated before assignment happens, so there is no danger of so-called aliasing in array, vector, or matrix operations. Contrast the behavior of the assignments to `u` and `x`, which start with the same values.

The loop assigning to `u` and the compound slicing assigning to `x`.

the following trivial Stan program.

```
transformed data {
  vector[4] x = [ 1, 2, 3, 4 ]';
  vector[4] u = [ 1, 2, 3, 4 ]';

  for (t in 2:4)
    u[t] = u[t - 1] * 3;

  x[2:4] = x[1:3] * 3;
```

```
  print("u = ", u);
  print("x = ", x);
}
```

The output it produces is,

```
u = [1,3,9,27]
x = [1,3,6,9]
```

In the loop version assigning to `u`, the values are updated before being used to define subsequent values; in the sliced expression assigning to `x`, the entire right-hand side is evaluated before assigning to the left-hand side.

17. Multiple Indexing and Range Indexing

Stan allows multiple indexes to be provided for containers (i.e., arrays, vectors, and matrices) in a single position, using either an array of integer indexes or range bounds. This allows many models to be vectorized. For instance, consider the likelihood for a varying-slope, varying-intercept hierarchical linear regression, which could be coded as

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

With multiple indexing, this can be coded in one line, leading to more efficient vectorized code.

```
y ~ normal(alpha[ii] + rows_dot_product(beta[ii], x), sigma);
```

This latter version is equivalent in speed to the clunky assignment to a local variable.

```
{
  vector[N] mu;
  for (n in 1:N)
    mu[n] = alpha[ii[n]] + beta[ii[n]] * x[n];
  y ~ normal(mu, sigma);
}
```

17.1. Multiple Indexing

The following is the simplest concrete example of multiple indexing with an array of integers; the ellipses stand for code defining the variables as indicated in the comments.

```
int c[3];
...             // define: c == (5, 9, 7)
int idxs[4];
...             // define: idxs == (3, 3, 1, 2)
int d[4];
d = c[idxs];    // result: d == (7, 7, 5, 9)
```

In general, the multiple indexed expression `c[idxs]` is defined as follows, assuming `idxs` is of size `K`.

```
c[idxs] = ( c[idxs[1]], c[idxs[2]], ..., c[idxs[K]] )
```

Thus `c[idxs]` is of the same size as `idxs`, which is `K` in this example.

Multiple indexing can also be used with multi-dimensional arrays. For example, consider the following.

```
int c[2, 3];
...              // define: c = ((1, 3, 5), ((7, 11, 13))
int idxs[4];
...              // define: idxs = (2, 2, 1, 2)
int d[4, 3]
d = c[idxs];     // result: d = ((7, 11, 13), (7, 11, 13),
                 //               (1, 3, 5), (7, 11, 13))
```

That is, putting an index in the first position acts exactly the same way as defined above. The fact that the values are themselves arrays makes no difference—the result is still defined by `c[idxs][j] == c[idxs[j]]`.

Multiple indexing may also be used in the second position of a multi-dimensional array. Continuing the above example, consider a single index in the first position and a multiple index in the second.

```
int e[4];
e = c[2, idxs]; // result:  c[2] = (7, 11, 13)
                // result:  e = (11, 11, 7, 11)
```

The single index is applied, the one-dimensional result is determined, then the multiple index is applied to the result. That is, `c[2,idxs]` evaluates to the same value as `c[2][idxs]`.

Multiple indexing can apply to more than one position of a multi-dimensional array. For instance, consider the following

```
int c[2, 3];
...                    // define: c = ((1, 3, 5), (7, 11, 13))
int idxs1[3];
...                    // define: idxs1 = (2, 2, 1)
int idxs2[2];
...                    // define: idxs2 = (1, 3)
int d[3, 2];
d = c[idxs1, idxs2];  // result: d = ((7, 13), (7, 13), (1, 5))
```

With multiple indexes, we no longer have `c[idxs1, idxs2]` being the same as

`c[idxs1][idxs2]`. Rather, the entry `d[i, j]` after executing the above is given by

```
d[i, j] == c[idxs1, idxs2][i, j] = c[idxs1[i], idxs2[j]]
```

This example illustrates the operation of multiple indexing in the general case: a multiple index like `idxs1` converts an index `i` used on the result (here, `c[idxs1, idxs2]`) to index `idxs1[i]` in the variable being indexed (here, `c`). In contrast, a single index just returns the value at that index, thus reducing dimensionality by one in the result.

17.2. Slicing with Range Indexes

Slicing returns a contiguous slice of a one-dimensional array, a contiguous sub-block of a two-dimensional array, and so on. Semantically, it is just a special form of multiple indexing.

Lower and Upper Bound Indexes

For instance, consider supplying an upper and lower bound for an index.

```
int c[7];
...
int d[4];
d = c[3:6];  // result: d == (c[3], c[4], c[5], c[6])
```

The range index `3:6` behaves semantically just like the multiple index `(3, 4, 5, 6)`. In terms of implementation, the sliced upper and/or lower bounded indices are faster and use less memory because they do not explicitly create a multiple index, but rather use a direct loop. They are also easier to read, so should be preferred over multiple indexes where applicable.

Lower or Upper Bound Indexes

It is also possible to supply just a lower bound, or just an upper bound. Writing `c[3:]` is just shorthand for `c[3:size(c)]`. Writing `c[:5]` is just shorthand for `c[1:5]`.

Full Range Indexes

Finally, it is possible to write a range index that covers the entire range of an array, either by including just the range symbol `(:)` as the index or leaving the index position empty. In both cases, `c[]` and `c[:]` are equal to `c[1:size(c)]`, which in turn is just equal to `c`.

17.3. Multiple Indexing on the Left of Assignments

Multiple expressions may be used on the left-hand side of an assignment statement, where they work exactly the same way as on the right-hand side in terms of picking out entries of a container. For example, consider the following.

```
int a[3];
int c[2];
int idxs[2];
...             // define: a == (1, 2, 3);  c == (5, 9)
                //         idxs = (3,2)
a[idxs] = c;    // result: a == (1, 9, 5)
```

The result above can be worked out by noting that the assignment sets `a[idxs[1]]` (`a[3]`) to `c[1]` (`5`) and `a[idxs[2]]` (`a[2]`) to `c[2]` (`9`).

The same principle applies when there are many multiple indexes, as in the following example.

```
int a[5, 7];
int c[2, 2];
...
a[2:3, 5:6] = c;  // result: a[2, 5] == c[1, 1];  a[2, 6] == c[1, 2]
                  //         a[3, 5] == c[2, 1];  a[3, 6] == c[2, 2]
```

As in the one-dimensional case, the right-hand side is written into the slice, block, or general chunk picked out by the left-hand side.

Usage on the left-hand side allows the full generality of multiple indexing, with single indexes reducing dimensionality and multiple indexes maintaining dimensionality while rearranging, slicing, or blocking. For example, it is valid to assign to a segment of a row of an array as follows.

```
int a[10, 13];
int c[2];
...
a[4, 2:3] = c;  // result:  a[4, 2] == c[1];  a[4, 3] == c[2]
```

Assign-by-Value and Aliasing

Aliasing issues arise when there are references to the same data structure on the right-hand and left-hand side of an assignment. For example, consider the array `a` in the following code fragment.

```
int a[3];
...                 // define: a == (5, 6, 7)
a[2:3] = a[1:2];
...                 // result: a == (5, 5, 6)
```

The reason the value of `a` after the assignment is $(5, 5, 6)$ rather than $(5, 5, 5)$ is that Stan behaves as if the right-hand side expression is evaluated to a fresh copy. As another example, consider the following.

```
int a[3];
int idxs[3];
...             // define idxs = (2, 1, 3)
a[idxs] = a;
```

In this case, it is evident why the right-hand side needs to be copied before the assignment.

It is tempting (but wrong) to think of the assignment `a[2:3] = a[1:2]` as executing the following assignments.

```
...                 // define: a = (5, 6, 7)
a[2] = a[1];        // result: a = (5, 5, 7)
a[3] = a[2];        // result: a = (5, 5, 5)!
```

This produces a different result than executing the assignment because `a[2]`'s value changes before it is used.

17.4. Multiple Indexes with Vectors and Matrices

Multiple indexes can be supplied to vectors and matrices as well as arrays of vectors and matrices.

Vectors

Vectors and row vectors behave exactly the same way as arrays with multiple indexes. If `v` is a vector, then `v[3]` is a scalar real value, whereas `v[2:4]` is a vector of size 3 containing the elements `v[2]`, `v[3]`, and `v[4]`.

The only subtlety with vectors is in inferring the return type when there are multiple indexes. For example, consider the following minimal example.

```
vector[5] v[3];
int idxs[7];
...
```

```
vector[7] u;
u = v[2, idxs];

real w[7];
w = v[idxs, 2];
```

The key is understanding that a single index always reduces dimensionality, whereas a multiple index never does. The dimensions with multiple indexes (and unindexed dimensions) determine the indexed expression's type. In the example above, because `v` is an array of vectors, `v[2, idxs]` reduces the array dimension but doesn't reduce the vector dimension, so the result is a vector. In contrast, `v[idxs, 2]` does not reduce the array dimension, but does reduce the vector dimension (to a scalar), so the result type for `w` is an array of reals. In both cases, the size of the multiple index (here, 7) determines the size of the result.

Matrices

Matrices are a bit trickier because they have two dimensions, but the underlying principle of type inference is the same—multiple indexes leave dimensions in place, whereas single indexes reduce them. The following code shows how this works for multiple indexing of matrices.

```
matrix[5,7] m;
...
row_vector[3] rv;
rv = m[4, 3:5];    // result is 1 x 3
...
vector[4] v;
v = m[2:5, 3];     // result is 3 x 1
...
matrix[3, 4] m2;
m2 = m[1:3, 2:5];  // result is 3 x 4
```

The key is realizing that any position with a multiple index or bounded index remains in play in the result, whereas any dimension with a single index is replaced with 1 in the resulting dimensions. Then the type of the result can be read off of the resulting dimensionality as indicated in the comments above.

Matrices with One Multiple Index

If matrices receive a single multiple index, the result is a matrix. So if `m` is a matrix, so is `m[2:4]`. In contrast, supplying a single index, `m[3]`, produces a row vector

result. That is, `m[3]` produces the same result as `m[3, ]` or `m[3, 1:cols(m)]`.

Arrays of Vectors or Matrices

With arrays of matrices, vectors, and row vectors, the basic access rules remain exactly the same: single indexes reduce dimensionality and multiple indexes redirect indexes. For example, consider the following example.

```
matrix[3, 4] m[5, 7];
...
matrix[3, 4] a[2];
a = m[1, 2:3];  // knock off first array dimension
a = m[3:4, 5];  // knock off second array dimension
```

In both assignments, the multiple index knocks off an array dimension, but it's different in both cases. In the first case, `a[i] == m[1, i + 1]`, whereas in the second case, `a[i] == m[i + 2, 5]`.

Continuing the previous example, consider the following.

```
...
vector[2] b;
b = a[1, 3, 2:3, 2];
```

Here, the two array dimensions are reduced as is the column dimension of the matrix, leaving only a row dimension index, hence the result is a vector. In this case, `b[j] == a[1, 3, 1 + j, 2]`.

This last example illustrates an important point: if there is a lower-bounded index, such as `2:3`, with lower bound 2, then the lower bound minus one is added to the index, as seen in the `1 + j` expression above.

Continuing further, consider continuing with the following.

```
...
row_vector[3] c[2];
c = a[4:5, 3, 1, 2: ];
```

Here, the first array dimension is reduced, leaving a single array dimension, and the row index of the matrix is reduced, leaving a row vector. For indexing, the values are given by `c[i, j] == a[i + 3, 3, 1, j + 1]`

17.5. Matrices with Parameters and Constants

Suppose you have a $3x3$ matrix and know that two entries are zero but the others are parameters. Such a situation arises in missing data situations and in problems with fixed structural parameters.

Suppose a 3×3 matrix is known to be zero at indexes $[1, 2]$ and $[1, 3]$. The indexes for parameters are included in a "melted" data-frame or database format.

```
transformed data {
  int<lower=1, upper=3> idxs[7, 2]
    = { {1, 1},
        {2, 1}, {2, 2}, {2, 3},
        {3, 1}, {3, 2}, {3, 3} };
  ...
```

The seven remaining parameters are declared as a vector.

```
parameters {
  vector[7] A_raw;
  ...
```

Then the full matrix `A` is constructed in the model block as a local variable.

```
model {
  matrix[3, 3] A;
  for (i in 1:7)
    A[idxs[i, 1], idxs[i, 2]] = A_raw[i];
  A[1, 2] = 0;
  A[1, 3] = 0;
  ...
```

This may seem like overkill in this setting, but in more general settings, the matrix size, vector size, and the `idxs` array will be too large to code directly. Similar techniques can be used to build up matrices with ad-hoc constraints, such as a handful of entries known to be positive.

18. User-Defined Functions

This chapter explains functions from a user perspective with examples; see the language reference for a full specification. User-defined functions allow computations to be encapsulated into a single named unit and invoked elsewhere by name. Similarly, functions allow complex procedures to be broken down into more understandable components. Writing modular code using descriptively named functions is easier to understand than a monolithic program, even if the latter is heavily commented.[1]

18.1. Basic Functions

Here's an example of a skeletal Stan program with a user-defined relative difference function employed in the generated quantities block to compute a relative differences between two parameters.

```
functions {
  real relative_diff(real x, real y) {
    real abs_diff;
    real avg_scale;
    abs_diff = fabs(x - y);
    avg_scale = (fabs(x) + fabs(y)) / 2;
    return abs_diff / avg_scale;
  }
}
...
generated quantities {
  real rdiff;
  rdiff = relative_diff(alpha, beta);
}
```

The function is named `relative_diff`, and is declared to have two real-valued arguments and return a real-valued result. It is used the same way a built-in function would be used in the generated quantities block.

[1] The main problem with comments is that they can be misleading, either due to misunderstandings on the programmer's part or because the program's behavior is modified after the comment is written. The program always behaves the way the code is written, which is why refactoring complex code into understandable units is preferable to simply adding comments.

User-Defined Functions Block

All functions are defined in their own block, which is labeled `functions` and must appear before all other program blocks. The user-defined functions block is optional.

Function Bodies

The body (the part between the curly braces) contains ordinary Stan code, including local variables. The new function is used in the generated quantities block just as any of Stan's built-in functions would be used.

Return Statements

Return statements, such as the one on the last line of the definition of `relative_diff` above, are only allowed in the bodies of function definitions. Return statements may appear anywhere in a function, but functions with non-void return types must end in a return statement.

Reject Statements

The Stan reject statement provides a mechanism to report errors or problematic values encountered during program execution. It accepts any number of quoted string literals or Stan expressions as arguments. This statement is typically embedded in a conditional statement in order to detect bad or illegal outcomes of some processing step.

Catching errors

Rejection is used to flag errors that arise in inputs or in program state. It is far better to fail early with a localized informative error message than to run into problems much further downstream (as in rejecting a state or failing to compute a derivative).

The most common errors that are coded is to test that all of the arguments to a function are legal. The following function takes a square root of its input, so requires non-negative inputs; it is coded to guard against illegal inputs.

```
real dbl_sqrt(real x) {
  if (!(x >= 0))
    reject("dblsqrt(x): x must be positive; found x = ", x);
  return 2 * sqrt(x);
}
```

The negation of the positive test is important, because it also catches the case where `x` is a not-a-number value. If the condition had been coded as `(x < 0)` it would not catch the not-a-number case, though it could be written as `(x < 0 || is_nan(x))`. The positive infinite case is allowed through, but could also be checked with the `is_inf(x)` function. The square root function does not itself reject, but some downstream consumer of `dbl_sqrt(-2)` would be likely to raise an error, at which point the origin of the illegal input requires detective work. Or even worse, as Matt Simpson pointed out in the GitHub comments, the function could go into an infinite loop if it starts with an infinite value and tries to reduce it by arithmetic, likely consuming all available memory and crashing an interface. Much better to catch errors early and report on their origin.

The effect of rejection depends on the program block in which the rejection is executed. In transformed data, rejections cause the program to fail to load. In transformed parameters or in the model block, rejections cause the current state to be rejected in the Metropolis sense.[2]

In generated quantities, rejections cause execution to halt because there is no way to recover and generate the remaining parameters, so extra care should be taken in calling functions in the generated quantities block.

Type Declarations for Functions

Function Argument	Local	Block
(unsized)	(unconstrained)	(constrained)
`int`	`int`	`int`
		`int<lower = L>`
		`int<upper = U>`
		`int<lower = L, upper = U>`
`real`	`real`	`real`
		`real<lower = L>`
		`real<upper = U>`
		`real<lower = L, upper = U>`
`vector`	`vector[N]`	`vector[N]`
		`vector[N]<lower = L>`
		`vector[N]<upper = U>`
		`vector[N]<lower = L, upper = U>`

[2] Just because this makes it possible to code a rejection sampler does not make it a good idea. Rejections break differentiability and the smooth exploration of the posterior. In Hamiltonian Monte Carlo, it can cause the sampler to be reduced to a diffusive random walk.

Function Argument	Local	Block
		`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>`
`matrix`	`matrix[M, N]`	`matrix[M, N]`
		`matrix[M, N]<lower = L>`
		`matrix[M, N]<upper = U>`
		`matrix[M, N]<lower = L, upper = U>`
	`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]`

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. Stan also allows arrays of any of these types, with slightly different declarations for function arguments and return types and variables.

Function argument and return types for vector and matrix types are not declared with their sizes, unlike type declarations for variables. Function argument type declarations may not be declared with constraints, either lower or upper bounds or structured constraints like forming a simplex or correlation matrix, (as is also the case for local variables); see the table of constrained types for full details.

For example, here's a function to compute the entropy of a categorical distribution

with simplex parameter `theta`.

```
real entropy(vector theta) {
  return sum(theta .* log(theta));
}
```

Although `theta` must be a simplex, only the type `vector` is used.[3]

Upper or lower bounds on values or constrained types are not allowed as return types or argument types in function declarations.

Array Types for Function Declarations

Array arguments have their own syntax, which follows that used in this manual for function signatures. For example, a function that operates on a two-dimensional array to produce a one-dimensional array might be declared as follows.

```
real[] baz(real[,] x);
```

The notation `[ ]` is used for one-dimensional arrays (as in the return above), `[ , ]` for two-dimensional arrays, `[ , , ]` for three-dimensional arrays, and so on.

Functions support arrays of any type, including matrix and vector types. As with other types, no constraints are allowed.

Data-only Function Arguments

A function argument which is a real-valued type or a container of a real-valued type, i.e., not an integer type or integer array type, can be qualified using the prefix qualifier `data`. The following is an example of a data-only function argument.

```
real foo(real y, data real mu) {
  return -0.5 * (y - mu)^2;
}
```

This qualifier restricts this argument to being invoked with expressions which consist only of data variables, transformed data variables, literals, and function calls. A data-only function argument cannot involve real variables declared in the parameters, transformed parameters, or model block. Attempts to invoke a function using an expression which contains parameter, transformed parameters, or model block variables as a data-only argument will result in an error message from the parser.

[3] A range of built-in validation routines is coming to Stan soon! Alternatively, the `reject` statement can be used to check constraints on the simplex.

Use of the `data` qualifier must be consistent between the forward declaration and the definition of a functions.

This qualifier should be used when writing functions that call the built-in ordinary differential equation (ODE) solvers, algebraic solvers, or map functions. These higher-order functions have strictly specified signatures where some arguments of are data only expressions. (See the ODE solver chapter for more usage details and the functions reference manual for full definitions.) When writing a function which calls the ODE or algebraic solver, arguments to that function which are passed into the call to the solver, either directly or indirectly, should have the `data` prefix qualifier. This allows for compile-time type checking and increases overall program understandability.

18.2. Functions as Statements

In some cases, it makes sense to have functions that do not return a value. For example, a routine to print the lower-triangular portion of a matrix can be defined as follows.

```
functions {
  void pretty_print_tri_lower(matrix x) {
    if (rows(x) == 0) {
      print("empty matrix");
      return;
    }
    print("rows=", rows(x), " cols=", cols(x));
    for (m in 1:rows(x))
      for (n in 1:m)
        print("[", m, ",", n, "]=", x[m, n]);
  }
}
```

The special symbol `void` is used as the return type. This is not a type itself in that there are no values of type `void`; it merely indicates the lack of a value. As such, return statements for void functions are not allowed to have arguments, as in the return statement in the body of the previous example.

Void functions applied to appropriately typed arguments may be used on their own as statements. For example, the pretty-print function defined above may be applied to a covariance matrix being defined in the transformed parameters block.

```
transformed parameters {
```

```
  cov_matrix[K] Sigma;
  ... code to set Sigma ...
  pretty_print_tri_lower(Sigma);
  ...
```

18.3. Functions Accessing the Log Probability Accumulator

Functions whose names end in `_lp` are allowed to use sampling statements and `target +=` statements; other functions are not. Because of this access, their use is restricted to the transformed parameters and model blocks.

Here is an example of a function to assign standard normal priors to a vector of coefficients, along with a center and scale, and return the translated and scaled coefficients; see the reparameterization section for more information on efficient non-centered parameterizations

```
functions {
  vector center_lp(vector beta_raw, real mu, real sigma) {
    beta_raw ~ std_normal();
    sigma ~ cauchy(0, 5);
    mu ~ cauchy(0, 2.5);
    return sigma * beta_raw + mu;
  }
  ...
}
parameters {
  vector[K] beta_raw;
  real mu_beta;
  real<lower=0> sigma_beta;
  ...
transformed parameters {
  vector[K] beta;
  ...
  beta = center_lp(beta_raw, mu_beta, sigma_beta);
  ...
```

18.4. Functions Acting as Random Number Generators

A user-specified function can be declared to act as a (pseudo) random number generator (PRNG) by giving it a name that ends in `_rng`. Giving a function a name that ends in `_rng` allows it to access built-in functions and user-defined functions

that end in _rng, which includes all the built-in PRNG functions. Only functions ending in _rng are able access the built-in PRNG functions. The use of functions ending in _rng must therefore be restricted to transformed data and generated quantities blocks like other PRNG functions; they may also be used in the bodies of other user-defined functions ending in _rng.

For example, the following function generates an $N \times K$ data matrix, the first column of which is filled with 1 values for the intercept and the remaining entries of which have values drawn from a standard normal PRNG.

```
matrix predictors_rng(int N, int K) {
  matrix[N, K] x;
  for (n in 1:N) {
    x[n, 1] = 1.0;  // intercept
    for (k in 2:K)
      x[n, k] = normal_rng(0, 1);
  }
  return x;
}
```

The following function defines a simulator for regression outcomes based on a data matrix x, coefficients beta, and noise scale sigma.

```
vector regression_rng(vector beta, matrix x, real sigma) {
  vector[rows(x)] y;
  vector[rows(x)] mu;
  mu = x * beta;
  for (n in 1:rows(x))
    y[n] = normal_rng(mu[n], sigma);
  return y;
}
```

These might be used in a generated quantity block to simulate some fake data from a fitted regression model as follows.

```
parameters {
  vector[K] beta;
  real<lower=0> sigma;
  ...
generated quantities {
  matrix[N_sim, K] x_sim;
  vector[N_sim] y_sim;
```

```
  x_sim = predictors_rng(N_sim, K);
  y_sim = regression_rng(beta, x_sim, sigma);
}
```

A more sophisticated simulation might fit a multivariate normal to the predictors `x` and use the resulting parameters to generate multivariate normal draws for `x_sim`.

18.5. User-Defined Probability Functions

Probability functions are distinguished in Stan by names ending in `_lpdf` for density functions and `_lpmf` for mass functions; in both cases, they must have `real` return types.

Suppose a model uses several standard normal distributions, for which there is not a specific overloaded density nor defaults in Stan. So rather than writing out the location of 0 and scale of 1 for all of them, a new density function may be defined and reused.

```
functions {
  real unit_normal_lpdf(real y) {
    return normal_lpdf(y | 0, 1);
  }
}
...
model {
  alpha ~ unit_normal();
  beta ~ unit_normal();
  ...
}
```

The ability to use the `unit_normal` function as a density is keyed off its name ending in `_lpdf` (names ending in `_lpmf` for probability mass functions work the same way).

In general, if `foo_lpdf` is defined to consume $N + 1$ arguments, then

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

can be used as shorthand for

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

As with the built-in functions, the suffix `_lpdf` is dropped and the first argument moves to the left of the sampling symbol (`~`) in the sampling statement.

Functions ending in `_lpmf` (for probability mass functions), behave exactly the same way. The difference is that the first argument of a density function (`_lpdf`) must be continuous (not an integer or integer array), whereas the first argument of a mass function (`_lpmf`) must be discrete (integer or integer array).

18.6. Overloading Functions

Stan does not permit overloading user-defined functions. This means that it is not possible to define two different functions with the same name, even if they have different signatures.

18.7. Documenting Functions

Functions will ideally be documented at their interface level. The Stan style guide for function documentation follows the same format as used by the Doxygen (C++) and Javadoc (Java) automatic documentation systems. Such specifications indicate the variables and their types and the return value, prefaced with some descriptive text.

For example, here's some documentation for the prediction matrix generator.

```
/**
 * Return a data matrix of specified size with rows
 * corresponding to items and the first column filled
 * with the value 1 to represent the intercept and the
 * remaining columns randomly filled with unit-normal draws.
 *
 * @param N Number of rows corresponding to data items
 * @param K Number of predictors, counting the intercept, per
 *          item.
 * @return Simulated predictor matrix.
 */
matrix predictors_rng(int N, int K) {
  ...
```

The comment begins with `/**`, ends with `*/`, and has an asterisk (`*`) on each line. It uses `@param` followed by the argument's identifier to document a function argument. The tag `@return` is used to indicate the return value. Stan does not (yet) have an automatic documentation generator like Javadoc or Doxygen, so this just looks like a big comment starting with `/*` and ending with `*/` to the Stan parser.

For functions that raise exceptions, exceptions can be documented using `@throws`.[4]

For example,

```
 ...
 * @param theta
 * @throws If any of the entries of theta is negative.
 */
real entropy(vector theta) {
  ...
```

Usually an exception type would be provided, but these are not exposed as part of the Stan language, so there is no need to document them.

18.8. Summary of Function Types

Functions may have a void or non-void return type and they may or may not have one of the special suffixes, `_lpdf`, `_lpmf`, `_lp`, or `_rng`.

Void vs. Non-Void Return

Only functions declared to return `void` may be used as statements. These are also the only functions that use `return` statements with no arguments.

Only functions declared to return non-`void` values may be used as expressions. These functions require `return` statements with arguments of a type that matches the declared return type.

Suffixed or Non-Suffixed

Only functions ending in `_lpmf` or `_lpdf` and with return type `real` may be used as probability functions in sampling statements.

Only functions ending in `_lp` may access the log probability accumulator through sampling statements or `target +=` statements. Such functions may only be used in the transformed parameters or model blocks.

Only functions ending in `_rng` may access the built-in pseudo-random number generators. Such functions may only be used in the generated quantities block or transformed data block, or in the bodies of other user-defined functions ending in `_rng`.

[4] As of Stan 2.9.0, the only way a user-defined producer will raise an exception is if a function it calls (including sampling statements) raises an exception via the reject statement.

18.9. Recursive Functions

Stan supports recursive function definitions, which can be useful for some applications. For instance, consider the matrix power operation, A^n, which is defined for a square matrix A and positive integer n by

$$A^n = \begin{cases} \mathrm{I} & \text{if } n = 0, \text{ and} \\ A\,A^{n-1} & \text{if } n > 0. \end{cases}$$

where I is the identity matrix. This definition can be directly translated to a recursive function definition.

```
matrix matrix_pow(matrix a, int n);

matrix matrix_pow(matrix a, int n) {
  if (n == 0)
    return diag_matrix(rep_vector(1, rows(a)));
  else
    return a *  matrix_pow(a, n - 1);
}
```

The forward declaration of the function signature before it is defined is necessary so that the embedded use of `matrix_pow` is well-defined when it is encountered. It would be more efficient to not allow the recursion to go all the way to the base case, adding the following conditional clause.

```
else if (n == 1)
  return a;
```

18.10. Truncated Random Number Generation

Generation with Inverse CDFs

To generate random numbers, it is often sufficient to invert their cumulative distribution functions. This is built into many of the random number generators. For example, to generate a standard logistic variate, first generate a uniform variate $u \sim \mathsf{uniform}(0, 1)$, then run through the inverse cumulative distribution function, $y = \mathrm{logit}(u)$. If this were not already built in as `logistic_rng(0, 1)`, it could be coded in Stan directly as

```
real standard_logistic_rng() {
  real u = uniform_rng(0, 1);
```

```
  real y = logit(u);
  return y;
}
```

Following the same pattern, a standard normal RNG could be coded as

```
real standard_normal_rng() {
  real u = uniform_rng(0, 1);
  real y = inv_Phi(u);
  return y;
}
```

that is, $y = \Phi^{-1}(u)$, where Φ^{-1} is the inverse cumulative distribution function for the standard normal distribution, implemented in the Stan function `inv_Phi`.

In order to generate non-standard variates of the location-scale variety, the variate is scaled by the scale parameter and shifted by the location parameter. For example, to generate $\mathsf{normal}(\mu, \sigma)$ variates, it is enough to generate a uniform variate $u \sim \mathsf{uniform}(0, 1)$, then convert it to a standard normal variate, $z = \Phi(u)$, where Φ is the inverse cumulative distribution function for the standard normal, and then, finally, scale and translate it, $y = \mu + \sigma \times z$. In code,

```
real my_normal_rng(real mu, real sigma) {
  real u = uniform_rng(0, 1);
  real z = inv_Phi(u);
  real y = mu + sigma * z;
  return y;
}
```

A robust version of this function would test that the arguments are finite and that `sigma` is non-negative, e.g.,

```
  if (is_nan(mu) || is_infinite(mu))
    reject("my_normal_rng: mu must be finite; ",
           "found mu = ", mu);
  if (is_nan(sigma) || is_infinite(sigma) || sigma < 0)
    reject("my_normal_rng: sigma must be finite and non-negative; ",
           "found sigma = ", sigma);
```

Truncated variate generation

Often truncated uniform variates are needed, as in survival analysis when a time of death is censored beyond the end of the observations. To generate a truncated

random variate, the cumulative distribution is used to find the truncation point in the inverse CDF, a uniform variate is generated in range, and then the inverse CDF translates it back.

Truncating below

For example, the following code generates a $\text{Weibull}(\alpha, \sigma)$ variate truncated below at a time t,[5]

```
real weibull_lb_rng(real alpha, real sigma, real t) {
  real p = weibull_cdf(lt, alpha, sigma);   // cdf for lb
  real u = uniform_rng(p, 1);               // unif in bounds
  real y = sigma * (-log1m(u))^inv(alpha);  // inverse cdf
  return y;
}
```

Truncating above and below

If there is a lower bound and upper bound, then the CDF trick is used twice to find a lower and upper bound. For example, to generate a $\text{normal}(\mu, \sigma)$ truncated to a region (a, b), the following code suffices,

```
real normal_lub_rng(real mu, real sigma, real lb, real ub) {
  real p_lb = normal_cdf(lb, mu, sigma);
  real p_ub = normal_cdf(ub, mu, sigma);
  real u = uniform_rng(p_lb, p_ub);
  real y = mu + sigma * inv_Phi(u);
  return y;
}
```

To make this more robust, all variables should be tested for finiteness, `sigma` should be tested for positiveness, and `lb` and `ub` should be tested to ensure the upper bound is greater than the lower bound. While it may be tempting to compress lines, the variable names serve as a kind of chunking of operations and naming for readability; compare the multiple statement version above with the single statement

```
  return mu + sigma * inv_Phi(uniform_rng(normal_cdf(lb, mu, sigma),
```

[5]The original code and impetus for including this in the manual came from the Stan forums post http://discourse.mc-stan.org/t/rng-for-truncated-distributions/3122/7; by user `lcomm`, who also explained truncation above and below.

```
                                        normal_cdf(ub, mu, sigma)));
```

for readability. The names like `p` indicate probabilities, and `p_lb` and `p_ub` indicate the probabilities of the bounds. The variable `u` is clearly named as a uniform variate, and `y` is used to denote the variate being generated itself.

19. Custom Probability Functions

Custom distributions may also be implemented directly within Stan's programming language. The only thing that is needed is to increment the total log probability. The rest of the chapter provides examples.

19.1. Examples

Triangle distribution

A simple example is the triangle distribution, whose density is shaped like an isosceles triangle with corners at specified bounds and height determined by the constraint that a density integrate to 1. If $\alpha \in \mathbb{R}$ and $\beta \in \mathbb{R}$ are the bounds, with $\alpha < \beta$, then $y \in (\alpha, \beta)$ has a density defined as follows.

$$\mathsf{triangle}(y \mid \alpha, \beta) = \frac{2}{\beta - \alpha} \left(1 - \left| y - \frac{\alpha + \beta}{\beta - \alpha} \right| \right)$$

If $\alpha = -1$, $\beta = 1$, and $y \in (-1, 1)$, this reduces to

$$\mathsf{triangle}(y \mid -1, 1) = 1 - |y|.$$

Consider the following Stan implementation of $\mathsf{triangle}(-1, 1)$ for sampling.

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

The single scalar parameter `y` is declared as lying in the interval `(-1,1)`. The total log probability is incremented with the joint log probability of all parameters, i.e., $\log \mathsf{Triangle}(y \mid -1, 1)$. This value is coded in Stan as `log1m(fabs(y))`. The function `log1m` is defined so that `log1m(x)` has the same value as $\log(1 - x)$, but the computation is faster, more accurate, and more stable.

The constrained type `real<lower=-1,upper=1>` declared for `y` is critical for correct sampling behavior. If the constraint on `y` is removed from the program, say by

declaring y as having the unconstrained scalar type `real`, the program would compile, but it would produce arithmetic exceptions at run time when the sampler explored values of y outside of $(-1, 1)$.

Now suppose the log probability function were extended to all of $\mathbb{R}$ as follows by defining the probability to be `log(0.0)`, i.e., $-\infty$, for values outside of $(-1, 1)$.

```
target += log(fmax(0.0,1 - fabs(y)));
```

With the constraint on y in place, this is just a less efficient, slower, and less arithmetically stable version of the original program. But if the constraint on y is removed, the model will compile and run without arithmetic errors, but will not sample properly.[1]

Exponential distribution

If Stan didn't happen to include the exponential distribution, it could be coded directly using the following assignment statement, where `lambda` is the inverse scale and y the sampled variate.

```
target += log(lambda) - y * lambda;
```

This encoding will work for any `lambda` and y; they can be parameters, data, or one of each, or even local variables.

The assignment statement in the previous paragraph generates C++ code that is similar to that generated by the following sampling statement.

```
y ~ exponential(lambda);
```

There are two notable differences. First, the sampling statement will check the inputs to make sure both `lambda` is positive and y is non-negative (which includes checking that neither is the special not-a-number value).

The second difference is that if `lambda` is not a parameter, transformed parameter, or local model variable, the sampling statement is clever enough to drop the `log(lambda)` term. This results in the same posterior because Stan only needs the log probability up to an additive constant. If `lambda` and y are both constants, the sampling statement will drop both terms (but still check for out-of-domain errors on the inputs).

[1] The problem is the (extremely!) light tails of the triangle distribution. The standard HMC and NUTS samplers can't get into the corners of the triangle properly. Because the Stan code declares y to be of type `real<lower = -1, upper = 1>`, the inverse logit transform is applied to the unconstrained variable and its log absolute derivative added to the log probability. The resulting distribution on the logit-transformed y is well behaved.

Bivariate normal cumulative distribution function

For another example of user-defined functions, consider the following definition of the bivariate normal cumulative distribution function (CDF) with location zero, unit variance, and correlation `rho`. That is, it computes

$$\texttt{binormal\_cdf}(z_1, z_2, \rho) = \mathsf{Pr}[Z_1 > z_1 \text{ and } Z_2 > z_2]$$

where the random 2-vector Z has the distribution

$$Z \sim \text{multivariate normal} \left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 & \rho \\ \rho & 1 \end{bmatrix}] \right).$$

The following Stan program implements this function,

```
real binormal_cdf(real z1, real z2, real rho) {
  if (z1 != 0 || z2 != 0) {
    real denom = fabs(rho) < 1.0 ? sqrt((1 + rho) * (1 - rho)) : not_a_numbe
    real a1 = (z2 / z1 - rho) / denom;
    real a2 = (z1 / z2 - rho) / denom;
    real product = z1 * z2;
    real delta = product < 0 || (product == 0 && (z1 + z2) < 0);
    return 0.5 * (Phi(z1) + Phi(z2) - delta) - owens_t(z1, a1) - owens_t(z2,
  }
  return 0.25 + asin(rho) / (2 * pi());
}
```

20. Problematic Posteriors

Mathematically speaking, with a proper posterior, one can do Bayesian inference and that's that. There is not even a need to require a finite variance or even a finite mean—all that's needed is a finite integral. Nevertheless, modeling is a tricky business and even experienced modelers sometimes code models that lead to improper priors. Furthermore, some posteriors are mathematically sound, but ill-behaved in practice. This chapter discusses issues in models that create problematic posterior inferences, either in general for Bayesian inference or in practice for Stan.

20.1. Collinearity of Predictors in Regressions

This section discusses problems related to the classical notion of identifiability, which lead to ridges in the posterior density and wreak havoc with both sampling and inference.

Examples of Collinearity

Redundant Intercepts

The first example of collinearity is an artificial example involving redundant intercept parameters.[1]

Suppose there are observations y_n for $n \in \{1, \ldots, N\}$, two intercept parameters λ_1 and λ_2, a scale parameter $\sigma > 0$, and the sampling distribution

$$y_n \sim \mathsf{normal}(\lambda_1 + \lambda_2, \sigma).$$

For any constant q, the sampling density for y does not change if we add q to λ_1 and subtract it from λ_2, i.e.,

$$p(y \mid \lambda_1, \lambda_2, \sigma) = p(y \mid \lambda_1 + q, \lambda_2 - q, \sigma).$$

The consequence is that an improper uniform prior $p(\mu, \sigma) \propto 1$ leads to an improper posterior. This impropriety arises because the neighborhoods around $\lambda_1 + q, \lambda_2 - q$

[1] This example was raised by Richard McElreath on the Stan users group in a query about the difference in behavior between Gibbs sampling as used in BUGS and JAGS and the Hamiltonian Monte Carlo (HMC) and no-U-turn samplers (NUTS) used by Stan.

have the same mass no matter what q is. Therefore, a sampler would need to spend as much time in the neighborhood of $\lambda_1 = 1\,000\,000\,000$ and $\lambda_2 = -1\,000\,000\,000$ as it does in the neighborhood of $\lambda_1 = 0$ and $\lambda_2 = 0$, and so on for ever more far-ranging values.

The marginal posterior $p(\lambda_1, \lambda_2 \mid y)$ for this model is thus improper.[2]

The impropriety shows up visually as a ridge in the posterior density, as illustrated in the left-hand plot. The ridge for this model is along the line where $\lambda_2 = \lambda_1 + c$ for some constant c.

Contrast this model with a simple regression with a single intercept parameter μ and sampling distribution

$$y_n \sim \mathsf{normal}(\mu, \sigma).$$

Even with an improper prior, the posterior is proper as long as there are at least two data points y_n with distinct values.

Ability and Difficulty in IRT Models

Consider an item-response theory model for students $j \in 1{:}J$ with abilities α_j and test items $i \in 1{:}I$ with difficulties β_i. The observed data are an $I \times J$ array with entries $y_{i,j} \in \{0, 1\}$ coded such that $y_{i,j} = 1$ indicates that student j answered question i correctly. The sampling distribution for the data is

$$y_{i,j} \sim \mathsf{Bernoulli}(\mathrm{logit}^{-1}(\alpha_j - \beta_i)).$$

For any constant c, the probability of y is unchanged by adding a constant c to all the abilities and subtracting it from all the difficulties, i.e.,

$$p(y \mid \alpha, \beta) = p(y \mid \alpha + c, \beta - c).$$

This leads to a multivariate version of the ridge displayed by the regression with two intercepts discussed above.

[2]The marginal posterior $p(\sigma \mid y)$ for σ is proper here as long as there are at least two distinct data points.

General Collinear Regression Predictors

The general form of the collinearity problem arises when predictors for a regression are collinear. For example, consider a linear regression sampling distribution

$$y_n \sim \mathsf{normal}(x_n\beta, \sigma)$$

for an N-dimensional observation vector y, an $N \times K$ predictor matrix x, and a K-dimensional coefficient vector β.

Now suppose that column k of the predictor matrix is a multiple of column k', i.e., there is some constant c such that $x_{n,k} = c\, x_{n,k'}$ for all n. In this case, the coefficients β_k and $\beta_{k'}$ can covary without changing the predictions, so that for any $d \neq 0$,

$$p(y \mid \ldots, \beta_k, \ldots, \beta_{k'}, \ldots, \sigma) = p(y \mid \ldots, d\beta_k, \ldots, \frac{d}{c}\,\beta_{k'}, \ldots, \sigma).$$

Even if columns of the predictor matrix are not exactly collinear as discussed above, they cause similar problems for inference if they are nearly collinear.

Multiplicative Issues with Discrimination in IRT

Consider adding a discrimination parameter δ_i for each question in an IRT model, with data sampling model

$$y_{i,j} \sim \mathsf{Bernoulli}(\mathrm{logit}^{-1}(\delta_i(\alpha_j - \beta_i))).$$

For any constant $c \neq 0$, multiplying δ by c and dividing α and β by c produces the same likelihood,

$$p(y \mid \delta, \alpha, \beta) = p(y \mid c\delta, \frac{1}{c}\alpha, \frac{1}{c}\beta).$$

If $c < 0$, this switches the signs of every component in α, β, and δ without changing the density.

Softmax with K vs. $K - 1$ Parameters

In order to parameterize a K-simplex (i.e., a K-vector with non-negative values that sum to one), only $K - 1$ parameters are necessary because the Kth is just one minus the sum of the first $K - 1$ parameters, so that if θ is a K-simplex,

$$\theta_K = 1 - \sum_{k=1}^{K-1} \theta_k.$$

The `softmax` function maps a K-vector α of linear predictors to a K-simplex $\theta = \texttt{softmax}(\alpha)$ by defining

$$\theta_k = \frac{\exp(\alpha_k)}{\sum_{k'=1}^{K} \exp(\alpha_{k'})}.$$

The `softmax` function is many-to-one, which leads to a lack of identifiability of the unconstrained parameters α. In particular, adding or subtracting a constant from each α_k produces the same simplex θ.

Mitigating the Invariances

All of the examples discussed in the previous section allow translation or scaling of parameters while leaving the data probability density invariant. These problems can be mitigated in several ways.

Removing Redundant Parameters or Predictors

In the case of the multiple intercepts, λ_1 and λ_2, the simplest solution is to remove the redundant intercept, resulting in a model with a single intercept parameter μ and sampling distribution $y_n \sim \text{normal}(\mu, \sigma)$. The same solution works for solving the problem with collinearity—just remove one of the columns of the predictor matrix x.

Pinning Parameters

The IRT model without a discrimination parameter can be fixed by pinning one of its parameters to a fixed value, typically 0. For example, the first student ability α_1 can be fixed to 0. Now all other student ability parameters can be interpreted as being relative to student 1. Similarly, the difficulty parameters are interpretable relative to student 1's ability to answer them.

This solution is not sufficient to deal with the multiplicative invariance introduced by the question discrimination parameters δ_i. To solve this problem, one of the difficulty parameters, say δ_1, must also be constrained. Because it's a multiplicative and not an additive invariance, it must be constrained to a non-zero value, with 1 being a convenient choice. Now all of the discrimination parameters may be interpreted relative to item 1's discrimination.

The many-to-one nature of $\texttt{softmax}(\alpha)$ is typically mitigated by pinning a component of α, for instance fixing $\alpha_K = 0$. The resulting mapping is one-to-one from $K - 1$ unconstrained parameters to a K-simplex. This is roughly how simplex-constrained

parameters are defined in Stan; see the reference manual chapter on constrained parameter transforms for a precise definition. The Stan code for creating a simplex from a $K-1$-vector can be written as

```
vector softmax_id(vector alpha) {
  vector[num_elements(alpha) + 1] alphac1;
  for (k in 1:num_elements(alpha))
    alphac1[k] = alpha[k];
  alphac1[num_elements(alphac1)] = 0;
  return softmax(alphac1);
}
```

Adding Priors

So far, the models have been discussed as if the priors on the parameters were improper uniform priors.

A more general Bayesian solution to these invariance problems is to impose proper priors on the parameters. This approach can be used to solve problems arising from either additive or multiplicative invariance.

For example, normal priors on the multiple intercepts,

$$\lambda_1, \lambda_2 \sim \mathsf{normal}(0, \tau),$$

with a constant scale τ, ensure that the posterior mode is located at a point where $\lambda_1 = \lambda_2$, because this minimizes $\log \mathsf{normal}(\lambda_1 \mid 0, \tau) + \log \mathsf{normal}(\lambda_2 \mid 0, \tau)$.[3]

The following plots show the posteriors for two intercept parameterization without prior, two intercept parameterization with standard normal prior, and one intercept reparameterization without prior. For all three cases, the posterior is plotted for 100 data points drawn from a standard normal.

The two intercept parameterization leads to an improper prior with a ridge extending infinitely to the northwest and southeast.

Adding a standard normal prior for the intercepts results in a proper posterior.

The single intercept parameterization with no prior also has a proper posterior.

[3] A Laplace prior (or an L1 regularizer for penalized maximum likelihood estimation) is not sufficient to remove this additive invariance. It provides shrinkage, but does not in and of itself identify the parameters because adding a constant to λ_1 and subtracting it from λ_2 results in the same value for the prior density.

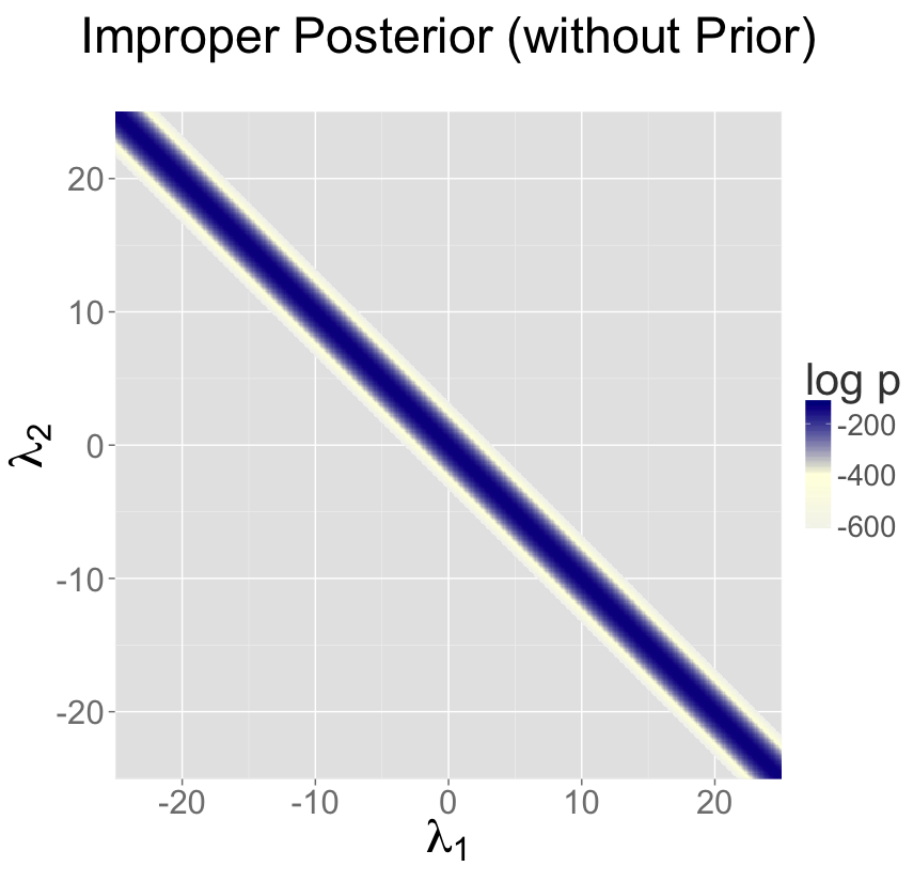

Figure 20.1: Two intercepts with improper prior

The addition of a prior to the two intercepts model is shown in the second plot; the final plot shows the result of reparameterizing to a single intercept.

An alternative strategy for identifying a K-simplex parameterization $\theta = \texttt{softmax}(\alpha)$ in terms of an unconstrained K-vector α is to place a prior on the components of α with a fixed location (that is, specifically avoid hierarchical priors with varying location). Unlike the approaching of pinning $\alpha_K = 0$, the prior-based approach models the K outcomes symmetrically rather than modeling $K-1$ outcomes relative to the K-th. The pinned parameterization, on the other hand, is usually more

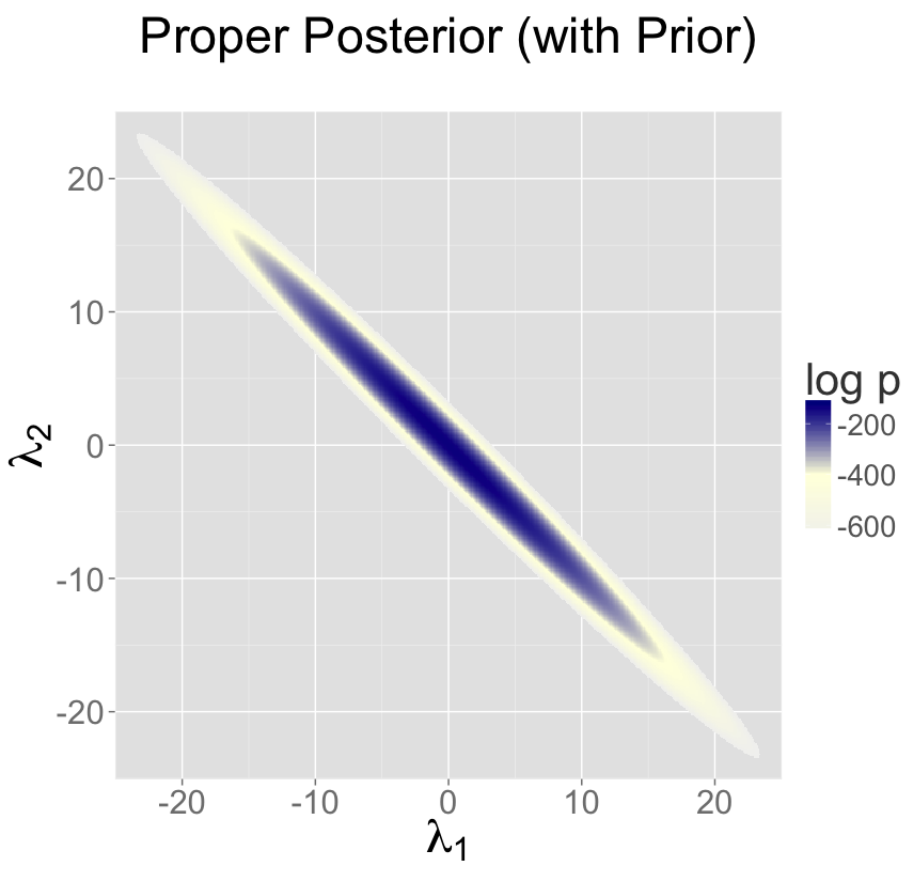

Figure 20.2: Two intercepts with proper prior

efficient statistically because it does not have the extra degree of (prior constrained) wiggle room.

Vague, Strongly Informative, and Weakly Informative Priors

Care must be used when adding a prior to resolve invariances. If the prior is taken to be too broad (i.e., too vague), the resolution is in theory only, and samplers will still struggle.

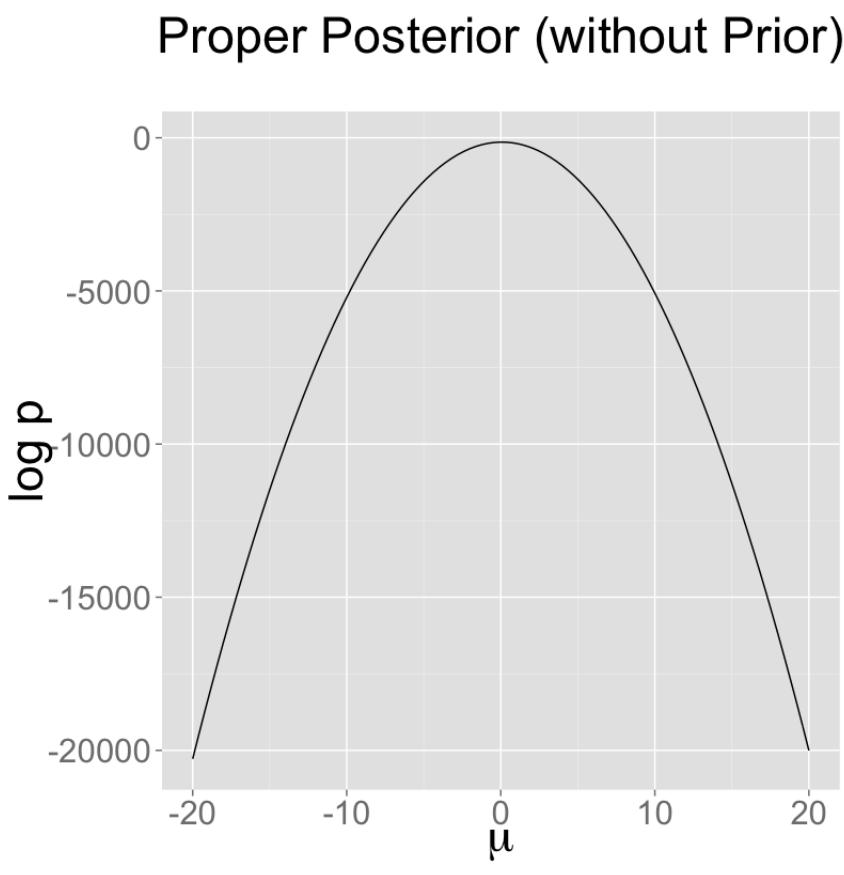

Figure 20.3: Single intercepts with improper prior

Ideally, a realistic prior will be formulated based on substantive knowledge of the problem being modeled. Such a prior can be chosen to have the appropriate strength based on prior knowledge. A strongly informative prior makes sense if there is strong prior information.

When there is not strong prior information, a weakly informative prior strikes the proper balance between controlling computational inference without dominating the data in the posterior. In most problems, the modeler will have at least some notion of the expected scale of the estimates and be able to choose a prior for identification

purposes that does not dominate the data, but provides sufficient computational control on the posterior.

Priors can also be used in the same way to control the additive invariance of the IRT model. A typical approach is to place a strong prior on student ability parameters α to control scale simply to control the additive invariance of the basic IRT model and the multiplicative invariance of the model extended with a item discrimination parameters; such a prior does not add any prior knowledge to the problem. Then a prior on item difficulty can be chosen that is either informative or weakly informative based on prior knowledge of the problem.

20.2. Label Switching in Mixture Models

Where collinearity in regression models can lead to infinitely many posterior maxima, swapping components in a mixture model leads to finitely many posterior maxima.

Mixture Models

Consider a normal mixture model with two location parameters μ_1 and μ_2, a shared scale $\sigma > 0$, a mixture ratio $\theta \in [0, 1]$, and likelihood

$$p(y \mid \theta, \mu_1, \mu_2, \sigma) = \prod_{n=1}^{N} \big(\theta \, \mathsf{normal}(y_n \mid \mu_1, \sigma) + (1 - \theta) \, \mathsf{normal}(y_n \mid \mu_2, \sigma)\big).$$

The issue here is exchangeability of the mixture components, because

$$p(\theta, \mu_1, \mu_2, \sigma \mid y) = p\big((1 - \theta), \mu_2, \mu_1, \sigma \mid y\big).$$

The problem is exacerbated as the number of mixture components K grows, as in clustering models, leading to $K!$ identical posterior maxima.

Convergence Monitoring and Effective Sample Size

The analysis of posterior convergence and effective sample size is also difficult for mixture models. For example, the $\hat{R}$ convergence statistic reported by Stan and the computation of effective sample size are both compromised by label switching. The problem is that the posterior mean, a key ingredient in these computations, is affected by label switching, resulting in a posterior mean for μ_1 that is equal to that of μ_2, and a posterior mean for θ that is always 1/2, no matter what the data are.

Some Inferences are Invariant

In some sense, the index (or label) of a mixture component is irrelevant. Posterior predictive inferences can still be carried out without identifying mixture components.

For example, the log probability of a new observation does not depend on the identities of the mixture components. The only sound Bayesian inferences in such models are those that are invariant to label switching. Posterior means for the parameters are meaningless because they are not invariant to label switching; for example, the posterior mean for θ in the two component mixture model will always be 1/2.

Highly Multimodal Posteriors

Theoretically, this should not present a problem for inference because all of the integrals involved in posterior predictive inference will be well behaved. The problem in practice is computation.

Being able to carry out such invariant inferences in practice is an altogether different matter. It is almost always intractable to find even a single posterior mode, much less balance the exploration of the neighborhoods of multiple local maxima according to the probability masses. In Gibbs sampling, it is unlikely for μ_1 to move to a new mode when sampled conditioned on the current values of μ_2 and θ. For HMC and NUTS, the problem is that the sampler gets stuck in one of the two "bowls" around the modes and cannot gather enough energy from random momentum assignment to move from one mode to another.

Even with a proper posterior, all known sampling and inference techniques are notoriously ineffective when the number of modes grows super-exponentially as it does for mixture models with increasing numbers of components.

Hacks as Fixes

Several hacks (i.e., "tricks") have been suggested and employed to deal with the problems posed by label switching in practice.

Parameter Ordering Constraints

One common strategy is to impose a constraint on the parameters that identifies the components. For instance, we might consider constraining $\mu_1 < \mu_2$ in the two-component normal mixture model discussed above. A problem that can arise from such an approach is when there is substantial probability mass for the opposite ordering $\mu_1 > \mu_2$. In these cases, the posteriors are affected by the constraint and true posterior uncertainty in μ_1 and μ_2 is not captured by the model with the constraint. In addition, standard approaches to posterior inference for event

probabilities is compromised. For instance, attempting to use M posterior samples to estimate $\Pr[\mu_1 > \mu_2]$, will fail, because the estimator

$$\Pr[\mu_1 > \mu_2] \approx \sum_{m=1}^{M} \mathrm{I}\left(\mu_1^{(m)} > \mu_2^{(m)}\right)$$

will result in an estimate of 0 because the posterior respects the constraint in the model.

Initialization around a Single Mode

Another common approach is to run a single chain or to initialize the parameters near realistic values.[4]

This can work better than the hard constraint approach if reasonable initial values can be found and the labels do not switch within a Markov chain. The result is that all chains are glued to a neighborhood of a particular mode in the posterior.

20.3. Component Collapsing in Mixture Models

It is possible for two mixture components in a mixture model to collapse to the same values during sampling or optimization. For example, a mixture of K normals might devolve to have $\mu_i = \mu_j$ and $\sigma_i = \sigma_j$ for $i \neq j$.

This will typically happen early in sampling due to initialization in MCMC or optimization or arise from random movement during MCMC. Once the parameters match for a given draw (m), it can become hard to escape because there can be a trough of low-density mass between the current parameter values and the ones without collapsed components.

It may help to use a smaller step size during warmup, a stronger prior on each mixture component's membership responsibility. A more extreme measure is to include additional mixture components to deal with the possibility that some of them may collapse.

In general, it is difficult to recover exactly the right K mixture components in a mixture model as K increases beyond one (yes, even a two-component mixture can have this problem).

[4] Tempering methods may be viewed as automated ways to carry out such a search for modes, though most MCMC tempering methods continue to search for modes on an ongoing basis; see (Swendsen and Wang 1986; Neal 1996b).

20.4. Posteriors with Unbounded Densities

In some cases, the posterior density grows without bounds as parameters approach certain poles or boundaries. In such, there are no posterior modes and numerical stability issues can arise as sampled parameters approach constraint boundaries.

Mixture Models with Varying Scales

One such example is a binary mixture model with scales varying by component, σ_1 and σ_2 for locations μ_1 and μ_2. In this situation, the density grows without bound as $\sigma_1 \to 0$ and $\mu_1 \to y_n$ for some n; that is, one of the mixture components concentrates all of its mass around a single data item y_n.

Beta-Binomial Models with Skewed Data and Weak Priors

Another example of unbounded densities arises with a posterior such as $\mathsf{beta}(\phi \mid 0.5, 0.5)$, which can arise if seemingly weak beta priors are used for groups that have no data. This density is unbounded as $\phi \to 0$ and $\phi \to 1$. Similarly, a Bernoulli likelihood model coupled with a "weak" beta prior, leads to a posterior

$$
\begin{aligned}
p(\phi \mid y) &\propto \mathsf{beta}(\phi \mid 0.5, 0.5) \times \prod_{n=1}^{N} \mathsf{Bernoulli}(y_n \mid \phi) \\
&= \mathsf{beta}\left(\phi \,\middle|\, 0.5 + \sum_{n=1}^{N} y_n, 0.5 + N - \sum_{n=1}^{N} y_n\right).
\end{aligned}
$$

If $N = 9$ and each $y_n = 1$, the posterior is $\mathsf{beta}(\phi \mid 9.5, 0, 5)$. This posterior is unbounded as $\phi \to 1$. Nevertheless, the posterior is proper, and although there is no posterior mode, the posterior mean is well-defined with a value of exactly 0.95.

Constrained vs. Unconstrained Scales

Stan does not sample directly on the constrained $(0, 1)$ space for this problem, so it doesn't directly deal with unconstrained density values. Rather, the probability values ϕ are logit-transformed to $(-\infty, \infty)$. The boundaries at 0 and 1 are pushed out to $-\infty$ and ∞ respectively. The Jacobian adjustment that Stan automatically applies ensures the unconstrained density is proper. The adjustment for the particular case of $(0, 1)$ is $\log \mathrm{logit}^{-1}(\phi) + \log \mathrm{logit}(1 - \phi)$.

There are two problems that still arise, though. The first is that if the posterior mass for ϕ is near one of the boundaries, the logit-transformed parameter will have to sweep out long paths and thus can dominate the U-turn condition imposed by the no-U-turn sampler (NUTS). The second issue is that the inverse transform from the unconstrained space to the constrained space can underflow to 0 or overflow to 1, even when the unconstrained parameter is not infinite. Similar problems arise for the expectation terms in logistic regression, which is why the logit-scale parameterizations of the Bernoulli and binomial distributions are more stable.

20.5. Posteriors with Unbounded Parameters

In some cases, the posterior density will not grow without bound, but parameters will grow without bound with gradually increasing density values. Like the models discussed in the previous section that have densities that grow without bound, such models also have no posterior modes.

Separability in Logistic Regression

Consider a logistic regression model with N observed outcomes $y_n \in \{0, 1\}$, an $N \times K$ matrix x of predictors, a K-dimensional coefficient vector β, and sampling distribution

$$y_n \sim \mathsf{Bernoulli}(\mathrm{logit}^{-1}(x_n\beta)).$$

Now suppose that column k of the predictor matrix is such that $x_{n,k} > 0$ if and only if $y_n = 1$, a condition known as "separability." In this case, predictive accuracy on the observed data continue to improve as $\beta_k \to \infty$, because for cases with $y_n = 1$, $x_n\beta \to \infty$ and hence $\mathrm{logit}^{-1}(x_n\beta) \to 1$.

With separability, there is no maximum to the likelihood and hence no maximum likelihood estimate. From the Bayesian perspective, the posterior is improper and therefore the marginal posterior mean for β_k is also not defined. The usual solution to this problem in Bayesian models is to include a proper prior for β, which ensures a proper posterior.

20.6. Uniform Posteriors

Suppose your model includes a parameter ψ that is defined on $[0, 1]$ and is given a flat prior $\mathsf{uniform}(\psi \mid 0, 1)$. Now if the data don't tell us anything about ψ, the posterior is also $\mathsf{uniform}(\psi \mid 0, 1)$.

Although there is no maximum likelihood estimate for ψ, the posterior is uniform over a closed interval and hence proper. In the case of a uniform posterior on

$[0, 1]$, the posterior mean for ψ is well-defined with value $1/2$. Although there is no posterior mode, posterior predictive inference may nevertheless do the right thing by simply integrating (i.e., averaging) over the predictions for ψ at all points in $[0, 1]$.

20.7. Sampling Difficulties with Problematic Priors

With an improper posterior, it is theoretically impossible to properly explore the posterior. However, Gibbs sampling as performed by BUGS and JAGS, although still unable to properly sample from such an improper posterior, behaves differently in practice than the Hamiltonian Monte Carlo sampling performed by Stan when faced with an example such as the two intercept model discussed in the collinearity section and illustrated in the non-identifiable density plot.

Gibbs Sampling

Gibbs sampling, as performed by BUGS and JAGS, may appear to be efficient and well behaved for this unidentified model, but as discussed in the previous subsection, will not actually explore the posterior properly.

Consider what happens with initial values $\lambda_1^{(0)}, \lambda_2^{(0)}$. Gibbs sampling proceeds in iteration m by drawing

$$\begin{aligned}
\lambda_1^{(m)} &\sim p(\lambda_1 \mid \lambda_2^{(m-1)}, \sigma^{(m-1)}, y) \\
\lambda_2^{(m)} &\sim p(\lambda_2 \mid \lambda_1^{(m)}, \sigma^{(m-1)}, y) \\
\sigma^{(m)} &\sim p(\sigma \mid \lambda_1^{(m)}, \lambda_2^{(m)}, y).
\end{aligned}$$

Now consider the draw for λ_1 (the draw for λ_2 is symmetric), which is conjugate in this model and thus can be done efficiently. In this model, the range from which the next λ_1 can be drawn is highly constrained by the current values of λ_2 and σ. Gibbs will run quickly and provide seemingly reasonable inferences for $\lambda_1 + \lambda_2$. But it will not explore the full range of the posterior; it will merely take a slow random walk from the initial values. This random walk behavior is typical of Gibbs sampling when posteriors are highly correlated and the primary reason to prefer Hamiltonian Monte Carlo to Gibbs sampling for models with parameters correlated in the posterior.

Hamiltonian Monte Carlo Sampling

Hamiltonian Monte Carlo (HMC), as performed by Stan, is much more efficient at exploring posteriors in models where parameters are correlated in the posterior. In

this particular example, the Hamiltonian dynamics (i.e., the motion of a fictitious particle given random momentum in the field defined by the negative log posterior) is going to run up and down along the valley defined by the potential energy (ridges in log posteriors correspond to valleys in potential energy). In practice, even with a random momentum for λ_1 and λ_2, the gradient of the log posterior is going to adjust for the correlation and the simulation will run λ_1 and λ_2 in opposite directions along the valley corresponding to the ridge in the posterior log density.

No-U-Turn Sampling

Stan's default no-U-turn sampler (NUTS), is even more efficient at exploring the posterior; see Hoffman and Gelman (2011) and Hoffman and Gelman (2014). NUTS simulates the motion of the fictitious particle representing the parameter values until it makes a U-turn, it will be defeated in most cases, as it will just move down the potential energy valley indefinitely without making a U-turn. What happens in practice is that the maximum number of leapfrog steps in the simulation will be hit in many of the iterations, causing a large number of log probability and gradient evaluations (1000 if the max tree depth is set to 10, as in the default). Thus sampling will appear to be slow. This is indicative of an improper posterior, not a bug in the NUTS algorithm or its implementation. It is simply not possible to sample from an improper posterior! Thus the behavior of HMC in general and NUTS in particular should be reassuring in that it will clearly fail in cases of improper posteriors, resulting in a clean diagnostic of sweeping out large paths in the posterior.

Here are results of Stan runs with default parameters fit to $N = 100$ data points generated from $y_n \sim \text{normal}(0, 1)$:

Two Scale Parameters, Improper Prior

```
Inference for Stan model: improper_stan
Warmup took (2.7, 2.6, 2.9, 2.9) seconds, 11 seconds total
Sampling took (3.4, 3.7, 3.6, 3.4) seconds, 14 seconds total

                Mean     MCSE   StdDev        5%       95%  N_Eff  N_Eff/s
lp__        -5.3e+01  7.0e-02  8.5e-01  -5.5e+01  -5.3e+01    150       11
n_leapfrog__ 1.4e+03  1.7e+01  9.2e+02   3.0e+00   2.0e+03   2987      212
lambda1      1.3e+03  1.9e+03  2.7e+03  -2.3e+03   6.0e+03    2.1     0.15
lambda2     -1.3e+03  1.9e+03  2.7e+03  -6.0e+03   2.3e+03    2.1     0.15
sigma        1.0e+00  8.5e-03  6.2e-02   9.5e-01   1.2e+00     54      3.9
mu           1.6e-01  1.9e-03  1.0e-01  -8.3e-03   3.3e-01   2966      211
```

Two Scale Parameters, Weak Prior

```
Warmup took (0.40, 0.44, 0.40, 0.36) seconds, 1.6 seconds total
Sampling took (0.47, 0.40, 0.47, 0.39) seconds, 1.7 seconds total

                Mean     MCSE   StdDev        5%    95%  N_Eff  N_Eff/s  R_
lp__             -54  4.9e-02  1.3e+00  -5.7e+01    -53    728      421
n_leapfrog__     157  2.8e+00  1.5e+02   3.0e+00    511   3085     1784
lambda1         0.31  2.8e-01  7.1e+00  -1.2e+01     12    638      369
lambda2        -0.14  2.8e-01  7.1e+00  -1.2e+01     12    638      369
sigma            1.0  2.6e-03  8.0e-02   9.2e-01    1.2    939      543
mu              0.16  1.8e-03  1.0e-01  -8.1e-03   0.33   3289     1902
```

One Scale Parameter, Improper Prior

```
Warmup took (0.011, 0.012, 0.011, 0.011) seconds, 0.044 seconds total
Sampling took (0.017, 0.020, 0.020, 0.019) seconds, 0.077 seconds total

                Mean     MCSE  StdDev        5%   50%   95%  N_Eff  N_Eff/s
lp__             -54  2.5e-02    0.91  -5.5e+01   -53   -53   1318    17198
n_leapfrog__     3.2  2.7e-01     1.7   1.0e+00   3.0   7.0     39      507
mu              0.17  2.1e-03    0.10  -3.8e-03  0.17  0.33   2408    31417
sigma            1.0  1.6e-03   0.071   9.3e-01   1.0   1.2   2094    27321
```

On the top is the non-identified model with improper uniform priors and likelihood $y_n \sim \text{normal}(\lambda_1 + \lambda_2, \sigma)$.

In the middle is the same likelihood as the middle plus priors $\lambda_k \sim \text{normal}(0, 10)$.

On the bottom is an identified model with an improper prior, with likelihood $y_n \sim \text{normal}(\mu, \sigma)$. All models estimate μ at roughly 0.16 with low Monte Carlo standard error, but a high posterior standard deviation of 0.1; the true value $\mu = 0$ is within the 90% posterior intervals in all three models.

Examples: Fits in Stan

To illustrate the issues with sampling from non-identified and only weakly identified models, we fit three models with increasing degrees of identification of their parameters. The posteriors for these models is illustrated in the non-identifiable density plot. The first model is the unidentified model with two location parameters and no priors discussed in the collinearity section.

```
data {
  int N;
  real y[N];
```

```
}
parameters {
  real lambda1;
  real lambda2;
  real<lower=0> sigma;
}
transformed parameters {
  real mu;
  mu = lambda1 + lambda2;
}
model {
  y ~ normal(mu, sigma);
}
```

The second adds priors to the model block for `lambda1` and `lambda2` to the previous model.

```
lambda1 ~ normal(0, 10);
lambda2 ~ normal(0, 10);
```

The third involves a single location parameter, but no priors.

```
data {
  int N;
  real y[N];
}
parameters {
  real mu;
  real<lower=0> sigma;
}
model {
  y ~ normal(mu, sigma);
}
```

All three of the example models were fit in Stan 2.1.0 with default parameters (1000 warmup iterations, 1000 sampling iterations, NUTS sampler with max tree depth of 10). The results are shown in the non-identified fits figure. The key statistics from these outputs are the following.

- As indicated by `R_hat` column, all parameters have converged other than λ_1 and λ_2 in the non-identified model.
- The average number of leapfrog steps is roughly 3 in the identified model, 150

in the model identified by a weak prior, and 1400 in the non-identified model.

- The number of effective samples per second for μ is roughly 31,000 in the identified model, 1,900 in the model identified with weakly informative priors, and 200 in the non-identified model; the results are similar for σ.
- In the non-identified model, the 95% interval for λ_1 is (-2300,6000), whereas it is only (-12,12) in the model identified with weakly informative priors.
- In all three models, the simulated value of $\mu = 0$ and $\sigma = 1$ are well within the posterior 90% intervals.

The first two points, lack of convergence and hitting the maximum number of leapfrog steps (equivalently maximum tree depth) are indicative of improper posteriors. Thus rather than covering up the problem with poor sampling as may be done with Gibbs samplers, Hamiltonian Monte Carlo tries to explore the posterior and its failure is a clear indication that something is amiss in the model.

21. Reparameterization and Change of Variables

Stan supports a direct encoding of reparameterizations. Stan also supports changes of variables by directly incrementing the log probability accumulator with the log Jacobian of the transform.

21.1. Theoretical and Practical Background

A Bayesian posterior is technically a probability *measure*, which is a parameterization-invariant, abstract mathematical object.[1]

Stan's modeling language, on the other hand, defines a probability *density*, which is a non-unique, parameterization-dependent function in $\mathbb{R}^N \rightarrow \mathbb{R}^+$. In practice, this means a given model can be represented different ways in Stan, and different representations have different computational performances.

As pointed out by Gelman (2004) in a paper discussing the relation between parameterizations and Bayesian modeling, a change of parameterization often carries with it suggestions of how the model might change, because we tend to use certain natural classes of prior distributions. Thus, it's not *just* that we have a fixed distribution that we want to sample from, with reparameterizations being computational aids. In addition, once we reparameterize and add prior information, the model itself typically changes, often in useful ways.

21.2. Reparameterizations

Reparameterizations may be implemented directly using the transformed parameters block or just in the model block.

Beta and Dirichlet Priors

The beta and Dirichlet distributions may both be reparameterized from a vector of counts to use a mean and total count.

[1] This is in contrast to (penalized) maximum likelihood estimates, which are not parameterization invariant.

Beta Distribution

For example, the Beta distribution is parameterized by two positive count parameters $\alpha, \beta > 0$. The following example illustrates a hierarchical Stan model with a vector of parameters `theta` are drawn i.i.d. for a Beta distribution whose parameters are themselves drawn from a hyperprior distribution.

```
parameters {
  real<lower = 0> alpha;
  real<lower = 0> beta;
  ...
model {
  alpha ~ ...
  beta ~ ...
  for (n in 1:N)
    theta[n] ~ beta(alpha, beta);
  ...
```

It is often more natural to specify hyperpriors in terms of transformed parameters. In the case of the Beta, the obvious choice for reparameterization is in terms of a mean parameter

$$\phi = \alpha/(\alpha + \beta)$$

and total count parameter

$$\lambda = \alpha + \beta.$$

Following @[GelmanEtAl:2013, Chapter 5] the mean gets a uniform prior and the count parameter a Pareto prior with $p(\lambda) \propto \lambda^{-2.5}$.

```
parameters {
  real<lower=0,upper=1> phi;
  real<lower=0.1> lambda;
  ...
transformed parameters {
  real<lower=0> alpha = lambda * phi;
  real<lower=0> beta = lambda * (1 - phi);
  ...
model {
  phi ~ beta(1, 1); // uniform on phi, could drop
  lambda ~ pareto(0.1, 1.5);
  for (n in 1:N)
    theta[n] ~ beta(alpha, beta);
```

```
  ...
```

The new parameters, `phi` and `lambda`, are declared in the parameters block and the parameters for the Beta distribution, `alpha` and `beta`, are declared and defined in the transformed parameters block. And If their values are not of interest, they could instead be defined as local variables in the model as follows.

```
model {
  real alpha = lambda * phi
  real beta = lambda * (1 - phi);
...
  for (n in 1:N)
    theta[n] ~ beta(alpha, beta);
...
}
```

With vectorization, this could be expressed more compactly and efficiently as follows.

```
model {
  theta ~ beta(lambda * phi, lambda * (1 - phi));
...
}
```

If the variables `alpha` and `beta` are of interest, they can be defined in the transformed parameter block and then used in the model.

Jacobians not Necessary

Because the transformed parameters are being used, rather than given a distribution, there is no need to apply a Jacobian adjustment for the transform. For example, in the beta distribution example, `alpha` and `beta` have the correct posterior distribution.

Dirichlet Priors

The same thing can be done with a Dirichlet, replacing the mean for the Beta, which is a probability value, with a simplex. Assume there are $K > 0$ dimensions being considered ($K = 1$ is trivial and $K = 2$ reduces to the beta distribution case). The traditional prior is

```
parameters {
  vector[K] alpha;
```

```
  simplex[K] theta[N];
  ...
model {
  alpha ~ ...;
  for (n in 1:N)
    theta[n] ~ dirichlet(alpha);
}
```

This provides essentially K degrees of freedom, one for each dimension of `alpha`, and it is not obvious how to specify a reasonable prior for `alpha`.

An alternative coding is to use the mean, which is a simplex, and a total count.

```
parameters {
  simplex[K] phi;
  real<lower=0> kappa;
  simplex[K] theta[N];
  ...
transformed parameters {
  vector[K] alpha = kappa * phi;
  ...
}
model {
  phi ~ ...;
  kappa ~ ...;
  for (n in 1:N)
    theta[n] ~ dirichlet(alpha);
```

Now it is much easier to formulate priors, because `phi` is the expected value of `theta` and `kappa` (minus `K`) is the strength of the prior mean measured in number of prior observations.

Transforming Unconstrained Priors: Probit and Logit

If the variable u has a $\mathsf{uniform}(0,1)$ distribution, then $\mathrm{logit}(u)$ is distributed as $\mathsf{logistic}(0,1)$. This is because inverse logit is the cumulative distribution function (cdf) for the logistic distribution, so that the logit function itself is the inverse CDF and thus maps a uniform draw in $(0,1)$ to a logistically-distributed quantity.

Things work the same way for the probit case: if u has a $\mathsf{uniform}(0,1)$ distribution, then $\Phi^{-1}(u)$ has a $\mathsf{normal}(0,1)$ distribution. The other way around, if v has a $\mathsf{normal}(0,1)$ distribution, then $\Phi(v)$ has a $\mathsf{uniform}(0,1)$ distribution.

In order to use the probit and logistic as priors on variables constrained to $(0, 1)$, create an unconstrained variable and transform it appropriately. For comparison, the following Stan program fragment declares a $(0, 1)$-constrained parameter `theta` and gives it a beta prior, then uses it as a parameter in a distribution (here using `foo` as a placeholder).

```
parameters {
  real<lower = 0, upper = 1> theta;
...
model {
  theta ~ beta(a, b);
  ...
  y ~ foo(theta);
...
```

If the variables `a` and `b` are one, then this imposes a uniform distribution `theta`. If `a` and `b` are both less than one, then the density on `theta` has a U shape, whereas if they are both greater than one, the density of `theta` has an inverted-U or more bell-like shape.

Roughly the same result can be achieved with unbounded parameters that are probit or inverse-logit-transformed. For example,

```
parameters {
  real theta_raw;
...
transformed parameters {
  real<lower = 0, upper = 1> theta = inv_logit(theta_raw);
...
model {
  theta_raw ~ logistic(mu, sigma);
  ...
  y ~ foo(theta);
...
```

In this model, an unconstrained parameter `theta_raw` gets a logistic prior, and then the transformed parameter `theta` is defined to be the inverse logit of `theta_raw`. In this parameterization, `inv_logit(mu)` is the mean of the implied prior on `theta`. The prior distribution on `theta` will be flat if `sigma` is one and `mu` is zero, and will be U-shaped if `sigma` is larger than one and bell shaped if `sigma` is less than one.

When moving from a variable in $(0, 1)$ to a simplex, the same trick may be performed

using the softmax function, which is a multinomial generalization of the inverse logit function. First, consider a simplex parameter with a Dirichlet prior.

```
parameters {
  simplex[K] theta;
...
model {
  theta ~ dirichlet(a);
  ...
  y ~ foo(theta);
```

Now `a` is a vector with `K` rows, but it has the same shape properties as the pair `a` and `b` for a beta; the beta distribution is just the distribution of the first component of a Dirichlet with parameter vector $[ab]^\top$. To formulate an unconstrained prior, the exact same strategy works as for the beta.

```
parameters {
  vector[K] theta_raw;
...
transformed parameters {
  simplex[K] theta = softmax(theta_raw);
...
model {
  theta_raw ~ multi_normal_cholesky(mu, L_Sigma);
```

The multivariate normal is used for convenience and efficiency with its Cholesky-factor parameterization. Now the mean is controlled by `softmax(mu)`, but we have additional control of covariance through `L_Sigma` at the expense of having on the order of K^2 parameters in the prior rather than order K. If no covariance is desired, the number of parameters can be reduced back to K using a vectorized normal distribution as follows.

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

where either or both of `mu` and `sigma` can be vectors.

21.3. Changes of Variables

Changes of variables are applied when the transformation of a parameter is characterized by a distribution. The standard textbook example is the lognormal distribution, which is the distribution of a variable $y > 0$ whose logarithm $\log y$ has a normal distribution. The distribution is being assigned to $\log y$.

The change of variables requires an adjustment to the probability to account for the distortion caused by the transform. For this to work, univariate changes of variables must be monotonic and differentiable everywhere in their support.

For univariate changes of variables, the resulting probability must be scaled by the absolute derivative of the transform.

In the case of log normals, if y's logarithm is normal with mean μ and deviation σ, then the distribution of y is given by

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

Stan works on the log scale to prevent underflow, where

$$\log p(y) = \log \mathsf{normal}(\log y \mid \mu, \sigma) - \log y.$$

In Stan, the change of variables can be applied in the sampling statement. To adjust for the curvature, the log probability accumulator is incremented with the log absolute derivative of the transform. The lognormal distribution can thus be implemented directly in Stan as follows.[2]

```
parameters {
  real<lower=0> y;
  ...
model {
  log(y) ~ normal(mu, sigma);
  target += -log(y);
  ...
```

It is important, as always, to declare appropriate constraints on parameters; here `y` is constrained to be positive.

It would be slightly more efficient to define a local variable for the logarithm, as follows.

```
model {
  real log_y;
  log_y = log(y);
  log_y ~ normal(mu, sigma);
```

[2]This example is for illustrative purposes only; the recommended way to implement the lognormal distribution in Stan is with the built-in `lognormal` probability function; see the functions reference manual for details.

```
  target += -log_y;
  ...
```

If y were declared as data instead of as a parameter, then the adjustment can be ignored because the data will be constant and Stan only requires the log probability up to a constant.

Change of Variables vs. Transformations

This section illustrates the difference between a change of variables and a simple variable transformation. A transformation samples a parameter, then transforms it, whereas a change of variables transforms a parameter, then samples it. Only the latter requires a Jacobian adjustment.

It does not matter whether the probability function is expressed using a sampling statement, such as

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

or as an increment to the log probability function, as in

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

Gamma and Inverse Gamma Distribution

Like the log normal, the inverse gamma distribution is a distribution of variables whose inverse has a gamma distribution. This section contrasts two approaches, first with a transform, then with a change of variables.

The transform based approach to sampling `y_inv` with an inverse gamma distribution can be coded as follows.

```
parameters {
  real<lower=0> y;
}
transformed parameters {
  real<lower=0> y_inv;
  y_inv = 1 / y;
}
model {
  y ~ gamma(2,4);
}
```

The change-of-variables approach to sampling `y_inv` with an inverse gamma distribution can be coded as follows.

```
parameters {
  real<lower=0> y_inv;
}
transformed parameters {
  real<lower=0> y;
  y = 1 / y_inv;  // change variables
}
model {
  y ~ gamma(2,4);
  target +=  -2 * log(y_inv);  //  Jacobian adjustment;
}
```

The Jacobian adjustment is the log of the absolute derivative of the transform, which in this case is

$$\log \left| \frac{d}{du} \left(\frac{1}{u} \right) \right| = \log \left| -u^{-2} \right| = \log u^{-2} = -2 \log u.$$

Multivariate Changes of Variables

In the case of a multivariate transform, the log of the Jacobian of the transform must be added to the log probability accumulator. In Stan, this can be coded as follows in the general case where the Jacobian is not a full matrix.

```
parameters {
  vector[K] u;      // multivariate parameter
   ...
transformed parameters {
  vector[K] v;     // transformed parameter
  matrix[K, K] J;   // Jacobian matrix of transform
  ... compute v as a function of u ...
  ... compute J[m, n] = d.v[m] / d.u[n] ...
  target += log(fabs(determinant(J)));
  ...
model {
  v ~ ...;
  ...
```

If the Jacobian is known analytically, it will be more efficient to apply it directly than to call the determinant function, which is neither efficient nor particularly stable numerically.

In many cases, the Jacobian matrix will be triangular, so that only the diagonal elements will be required for the determinant calculation. Triangular Jacobians arise when each element `v[k]` of the transformed parameter vector only depends on elements `u[1]`, ..., `u[k]` of the parameter vector. For triangular matrices, the determinant is the product of the diagonal elements, so the transformed parameters block of the above model can be simplified and made more efficient by recoding as follows.

```
transformed parameters {
  ...
  vector[K] J_diag;  // diagonals of Jacobian matrix
  ...
  ... compute J[k, k] = d.v[k] / d.u[k] ...
  target += sum(log(J_diag));
  ...
```

21.4. Vectors with Varying Bounds

Stan only allows a single lower and upper bound to be declared in the constraints for a container data type. But suppose we have a vector of parameters and a vector of lower bounds? Then the transforms are calculated and their log Jacobians added to the log density accumulator; the Jacobian calculations are described in detail in the reference manual chapter on constrained parameter transforms.

Varying Lower Bounds

For example, suppose there is a vector parameter α with a vector L of lower bounds. The simplest way to deal with this if L is a constant is to shift a lower-bounded parameter.

```
data {
  int N;
  vector[N] L;  // lower bounds
  ...
parameters {
  vector<lower=0>[N] alpha_raw;
  ...
transformed parameters {
```

```
  vector[N] alpha = L + alpha_raw;
  ...
```

The Jacobian for adding a constant is one, so its log drops out of the log density.

Even if the lower bound is a parameter rather than data, there is no Jacobian required, because the transform from (L, α_{raw}) to $(L+\alpha_{\text{raw}}, \alpha_{\text{raw}})$ produces a Jacobian derivative matrix with a unit determinant.

It's also possible implement the transform by directly transforming an unconstrained parameter and accounting for the Jacobian.

```
data {
  int N;
  vector[N] L;  // lower bounds
  ...
parameters {
  vector[N] alpha_raw;
  ...
transformed parameters {
  vector[N] alpha = L + exp(alpha_raw);
  ...
model {
  target += sum(alpha_raw);  // log Jacobian
  ...
```

The adjustment in the the log Jacobian determinant of the transform mapping α_{raw} to $\alpha = L + \exp(\alpha_{\text{raw}})$. The details are simple in this case because the Jacobian is diagonal; see the reference manual chapter on constrained parameter transforms for full details. Here L can even be a vector containing parameters that don't depend on α_{raw}; if the bounds do depend on α_{raw} then a revised Jacobian needs to be calculated taking into account the dependencies.

Varying Upper and Lower Bounds

Suppose there are lower and upper bounds that vary by parameter. These can be applied to shift and rescale a parameter constrained to $(0, 1)$.

```
data {
  int N;
  vector[N] L;  // lower bounds
  vector[N] U;  // upper bounds
  ...
```

```
parameters {
  vector<lower=0, upper=1>[N] alpha_raw;
  ...
transformed parameters {
  vector[N] alpha = L + (U - L) .* alpha_raw;
```

The expression `U - L` is multiplied by `alpha_raw` elementwise to produce a vector of variables in $(0, U - L)$, then adding L results in a variable ranging between (L, U).

In this case, it is important that L and U are constants, otherwise a Jacobian would be required when multiplying by $U - L$.

22. Efficiency Tuning

This chapter provides a grab bag of techniques for optimizing Stan code, including vectorization, sufficient statistics, and conjugacy. At a coarse level, efficiency involves both the amount of time required for a computation and the amount of memory required. For practical applied statistical modeling, we are mainly concerned with reducing wall time (how long a program takes as measured by a clock on the wall) and keeping memory requirements within available bounds.

22.1. What is Efficiency?

The standard algorithm analyses in computer science measure efficiency asymptotically as a function of problem size (such as data, number of parameters, etc.) and typically do not consider constant additive factors like startup times or multiplicative factors like speed of operations. In practice, the constant factors are important; if run time can be cut in half or more, that's a huge gain. This chapter focuses on both the constant factors involved in efficiency (such as using built-in matrix operations as opposed to naive loops) and on asymptotic efficiency factors (such as using linear algorithms instead of quadratic algorithms in loops).

22.2. Efficiency for Probabilistic Models and Algorithms

Stan programs express models which are intrinsically statistical in nature. The algorithms applied to these models may or may not themselves be probabilistic. For example, given an initial value for parameters (which may itself be given deterministically or generated randomly), Stan's optimization algorithm (L-BFGS) for penalized maximum likelihood estimation is purely deterministic. Stan's sampling algorithms are based on Markov chain Monte Carlo algorithms, which are probabilistic by nature at every step. Stan's variational inference algorithm (ADVI) is probabilistic despite being an optimization algorithm; the randomization lies in a nested Monte Carlo calculation for an expected gradient.

With probabilistic algorithms, there will be variation in run times (and maybe memory usage) based on the randomization involved. For example, by starting too far out in the tail, iterative algorithms underneath the hood, such as the solvers for ordinary differential equations, may take different numbers of steps. Ideally this variation will be limited; when there is a lot of variation it can be a sign that

there is a problem with the model's parameterization in a Stan program or with initialization.

A well-behaved Stan program will have low variance between runs with different random initializations and differently seeded random number generators. But sometimes an algorithm can get stuck in one part of the posterior, typically due to high curvature. Such sticking almost always indicates the need to reparameterize the model. Just throwing away Markov chains with apparently poor behavior (slow, or stuck) can lead to bias in posterior estimates. This problem with getting stuck can often be overcome by lowering the initial step size to avoid getting stuck during adaptation and increasing the target acceptance rate in order to target a lower step size. This is because smaller step sizes allow Stan's gradient-based algorithms to better follow the curvature in the density or penalized maximum likelihood being fit.

22.3. Statistical vs. Computational Efficiency

There is a difference between pure computational efficiency and statistical efficiency for Stan programs fit with sampling-based algorithms. Computational efficiency measures the amount of time or memory required for a given step in a calculation, such as an evaluation of a log posterior or penalized likelihood.

Statistical efficiency typically involves requiring fewer steps in algorithms by making the statistical formulation of a model better behaved. The typical way to do this is by applying a change of variables (i.e., reparameterization) so that sampling algorithms mix better or optimization algorithms require less adaptation.

22.4. Model Conditioning and Curvature

Because Stan's algorithms (other than Riemannian Hamiltonian Monte Carlo) rely on step-based gradient-based approximations of the density (or penalized maximum likelihood) being fitted, posterior curvature not captured by this first-order approximation plays a central role in determining the statistical efficiency of Stan's algorithms.

A second-order approximation to curvature is provided by the Hessian, the matrix of second derivatives of the log density $\log p(\theta)$ with respect to the parameter vector θ, defined as

$$H(\theta) = \nabla \nabla \log p(\theta \mid y),$$

so that

$$H_{i,j}(\theta) = \frac{\partial^2 \log p(\theta \mid y)}{\partial \theta_i \, \partial \theta_j}.$$

For pure penalized maximum likelihood problems, the posterior log density $\log p(\theta \mid y)$ is replaced by the penalized likelihood function $\mathcal{L}(\theta) = \log p(y \mid \theta) - \lambda(\theta)$.

Condition Number and Adaptation

A good gauge of how difficult a problem the curvature presents is given by the condition number of the Hessian matrix H, which is the ratio of the largest to the smallest eigenvalue of H (assuming the Hessian is positive definite). This essentially measures the difference between the flattest direction of movement and the most curved. Typically, the step size of a gradient-based algorithm is bounded by the most sharply curved direction. With better conditioned log densities or penalized likelihood functions, it is easier for Stan's adaptation, especially the diagonal adaptations that are used as defaults.

Unit Scales without Correlation

Ideally, all parameters should be programmed so that they have unit scale and so that posterior correlation is reduced; together, these properties mean that there is no rotation or scaling required for optimal performance of Stan's algorithms. For Hamiltonian Monte Carlo, this implies a unit mass matrix, which requires no adaptation as it is where the algorithm initializes. Riemannian Hamiltonian Monte Carlo performs this conditioning on the fly at every step, but such conditioning is expensive computationally.

Varying Curvature

In all but very simple models (such as multivariate normals), the Hessian will vary as θ varies (an extreme example is Neal's funnel, as naturally arises in hierarchical models with little or no data). The more the curvature varies, the harder it is for all of the algorithms with fixed adaptation parameters (that is, everything but Riemannian Hamiltonian Monte Carlo) to find adaptations that cover the entire density well. Many of the variable transforms proposed are aimed at improving the conditioning of the Hessian and/or making it more consistent across the relevant portions of the density (or penalized maximum likelihood function) being fit.

For all of Stan's algorithms, the curvature along the path from the initial values of the parameters to the solution is relevant. For penalized maximum likelihood and variational inference, the solution of the iterative algorithm will be a single point, so this is all that matters. For sampling, the relevant "solution" is the typical set, which is the posterior volume where almost all draws from the posterior lies; thus, the typical set contains almost all of the posterior probability mass.

With sampling, the curvature may vary dramatically between the points on the path from the initialization point to the typical set and within the typical set. This is why adaptation needs to run long enough to visit enough points in the typical set to get a good first-order estimate of the curvature within the typical set. If adaptation is not run long enough, sampling within the typical set after adaptation will not be efficient. We generally recommend at least one hundred iterations after the typical set is reached (and the first effective draw is ready to be realized). Whether adaptation has run long enough can be measured by comparing the adaptation parameters derived from a set of diffuse initial parameter values.

Reparameterizing with a Change of Variables

Improving statistical efficiency is achieved by reparameterizing the model so that the same result may be calculated using a density or penalized maximum likelihood that is better conditioned. Again, see the example of reparameterizing Neal's funnel for an example, and also the examples in the change of variables chapter.

One has to be careful in using change-of-variables reparameterizations when using maximum likelihood estimation, because they can change the result if the Jacobian term is inadvertently included in the revised likelihood model.

22.5. Well-Specified Models

Model misspecification, which roughly speaking means using a model that doesn't match the data, can be a major source of slow code. This can be seen in cases where simulated data according to the model runs robustly and efficiently, whereas the real data for which it was intended runs slowly or may even have convergence and mixing issues. While some of the techniques recommended in the remaining sections of this chapter may mitigate the problem, the best remedy is a better model specification.

Counterintuitively, more complicated models often run faster than simpler models. One common pattern is with a group of parameters with a wide fixed prior such as `normal(0, 1000)`). This can fit slowly due to the mismatch between prior and posterior (the prior has support for values in the hundreds or even thousands, whereas the posterior may be concentrated near zero). In such cases, replacing the fixed prior with a hierarchical prior such as `normal(mu, sigma)`, where `mu` and `sigma` are new parameters with their own hyperpriors, can be beneficial.

22.6. Avoiding Validation

Stan validates all of its data structure constraints. For example, consider a transformed parameter defined to be a covariance matrix and then used as a covariance parameter in the model block.

```
transformed parameters {
  cov_matrix[K] Sigma;
  ...
}                                 // first validation
model {
  y ~ multi_normal(mu, Sigma);  // second validation
  ...
```

Because `Sigma` is declared to be a covariance matrix, it will be factored at the end of the transformed parameter block to ensure that it is positive definite. The multivariate normal log density function also validates that `Sigma` is positive definite. This test is expensive, having cubic run time (i.e., $\mathcal{O}(N^3)$ for $N \times N$ matrices), so it should not be done twice.

The test may be avoided by simply declaring `Sigma` to be a simple unconstrained matrix.

```
transformed parameters {
  matrix[K, K] Sigma;
  ...
model {
  y ~ multi_normal(mu, Sigma);  // only validation
```

Now the only validation is carried out by the multivariate normal.

22.7. Reparameterization

Stan's sampler can be slow in sampling from distributions with difficult posterior geometries. One way to speed up such models is through reparameterization. In some cases, reparameterization can dramatically increase effective sample size for the same number of iterations or even make programs that would not converge well behaved.

Example: Neal's Funnel

In this section, we discuss a general transform from a centered to a non-centered parameterization (Papaspiliopoulos, Roberts, and Sköld 2007).[1]

This reparameterization is helpful when there is not much data, because it separates the hierarchical parameters and lower-level parameters in the prior.

Neal (2003) defines a distribution that exemplifies the difficulties of sampling from some hierarchical models. Neal's example is fairly extreme, but can be trivially reparameterized in such a way as to make sampling straightforward. Neal's example has support for $y \in \mathbb{R}$ and $x \in \mathbb{R}^9$ with density

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

The probability contours are shaped like ten-dimensional funnels. The funnel's neck is particularly sharp because of the exponential function applied to y. A plot of the log marginal density of y and the first dimension x_1 is shown in the following plot.

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

The funnel can be implemented directly in Stan as follows.

```
parameters {
  real y;
  vector[9] x;
}
model {
  y ~ normal(0, 3);
  x ~ normal(0, exp(y/2));
}
```

When the model is expressed this way, Stan has trouble sampling from the neck of the funnel, where y is small and thus x is constrained to be near 0. This is due to the fact that the density's scale changes with y, so that a step size that works well in

[1] This parameterization came to be known on our mailing lists as the "Matt trick" after Matt Hoffman, who independently came up with it while fitting hierarchical models in Stan.

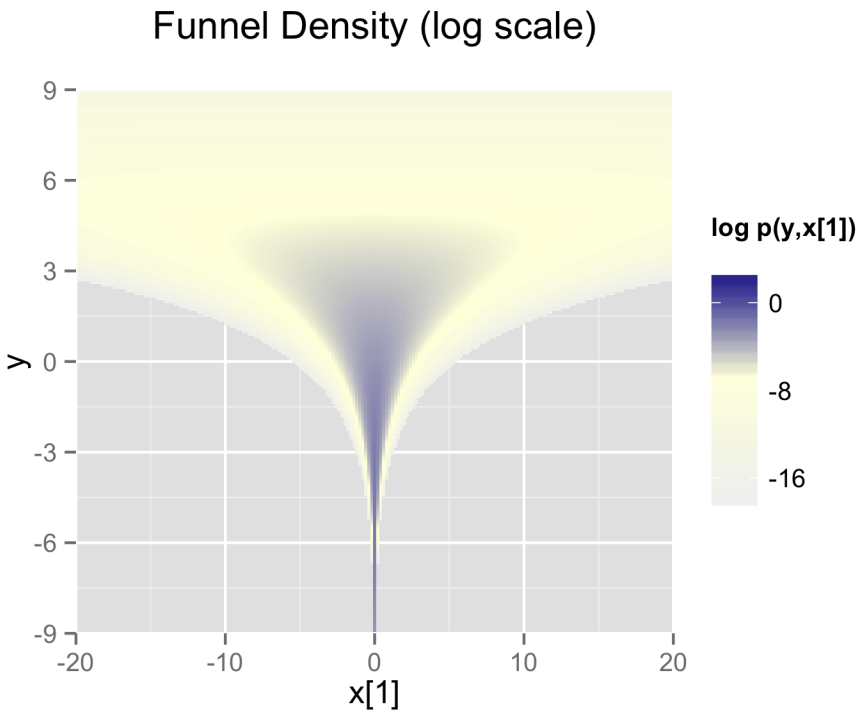

Figure 22.1: Neal's funnel density

the body will be too large for the neck, and a step size that works in the neck will be inefficient in the body. This can be seen in the following plot.

4000 draws are taken from a run of Stan's sampler with default settings. Both plots are restricted to the shown window of x_1 and y values; some draws fell outside of the displayed area as would be expected given the density. The samples are consistent with the marginal density $p(y) = \mathsf{normal}(y \mid 0, 3)$, which has mean 0 and standard deviation 3.

In this particular instance, because the analytic form of the density from which samples are drawn is known, the model can be converted to the following more efficient form.

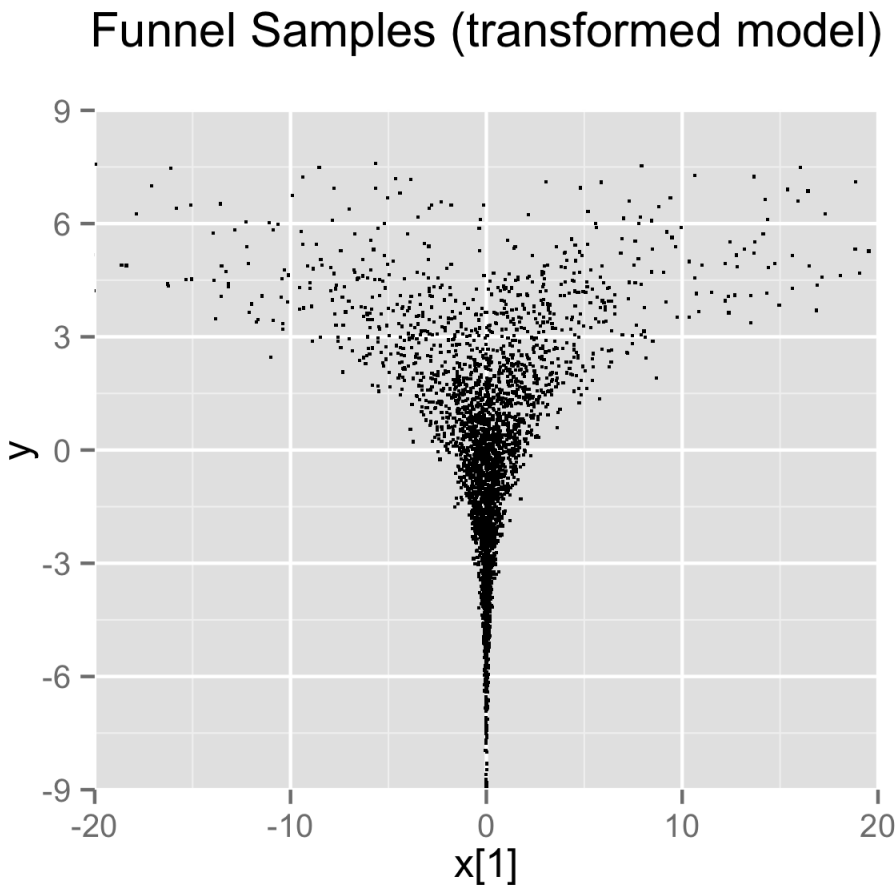

Figure 22.2:

```
parameters {
  real y_raw;
  vector[9] x_raw;
}
transformed parameters {
  real y;
  vector[9] x;

  y = 3.0 * y_raw;
  x = exp(y/2) * x_raw;
}
model {
  y_raw ~ std_normal(); // implies y ~ normal(0, 3)
  x_raw ~ std_normal(); // implies x ~ normal(0, exp(y/2))
}
```

In this second model, the parameters `x_raw` and `y_raw` are sampled as independent standard normals, which is easy for Stan. These are then transformed into samples from the funnel. In this case, the same transform may be used to define Monte Carlo samples directly based on independent standard normal samples; Markov chain Monte Carlo methods are not necessary. If such a reparameterization were used in Stan code, it is useful to provide a comment indicating what the distribution for the parameter implies for the distribution of the transformed parameter.

Reparameterizing the Cauchy

Sampling from heavy tailed distributions such as the Cauchy is difficult for Hamiltonian Monte Carlo, which operates within a Euclidean geometry.[2]

The practical problem is that tail of the Cauchy requires a relatively large step size compared to the trunk. With a small step size, the No-U-Turn sampler requires many steps when starting in the tail of the distribution; with a large step size, there will be too much rejection in the central portion of the distribution. This problem may be mitigated by defining the Cauchy-distributed variable as the transform of a uniformly distributed variable using the Cauchy inverse cumulative distribution function.

Suppose a random variable of interest X has a Cauchy distribution with location μ and scale τ, so that $X \sim \text{Cauchy}(\mu, \tau)$. The variable X has a cumulative distribution

[2] Riemannian Manifold Hamiltonian Monte Carlo (RMHMC) overcomes this difficulty by simulating the Hamiltonian dynamics in a space with a position-dependent metric; see Girolami and Calderhead (2011) and Betancourt (2012).

function $F_X : \mathbb{R} \to (0, 1)$ defined by

$$F_X(x) = \frac{1}{\pi} \arctan\left(\frac{x - \mu}{\tau}\right) + \frac{1}{2}.$$

The inverse of the cumulative distribution function, $F_X^{-1} : (0, 1) \to \mathbb{R}$, is thus

$$F_X^{-1}(y) = \mu + \tau \tan\left(\pi\left(y - \frac{1}{2}\right)\right).$$

Thus if the random variable Y has a unit uniform distribution, $Y \sim \text{uniform}(0, 1)$, then $F_X^{-1}(Y)$ has a Cauchy distribution with location μ and scale τ, i.e., $F_X^{-1}(Y) \sim \text{Cauchy}(\mu, \tau)$.

Consider a Stan program involving a Cauchy-distributed parameter `beta`.

```
parameters {
  real beta;
  ...
}
model {
  beta ~ cauchy(mu, tau);
  ...
}
```

This declaration of `beta` as a parameter may be replaced with a transformed parameter `beta` defined in terms of a uniform-distributed parameter `beta_unif`.

```
parameters {
  real<lower = -pi()/2, upper = pi()/2> beta_unif;
  ...
}
transformed parameters {
  real beta;
  beta = mu + tau * tan(beta_unif);  // beta ~ cauchy(mu, tau)
}
model {
  beta_unif ~ uniform(-pi()/2, pi()/2);  // not necessary
  ...
}
```

It is more convenient in Stan to transform a uniform variable on $(-\pi/2, \pi/2)$ than one on $(0, 1)$. The Cauchy location and scale parameters, `mu` and `tau`, may be defined

as data or may themselves be parameters. The variable `beta` could also be defined as a local variable if it does not need to be included in the sampler's output.

The uniform distribution on `beta_unif` is defined explicitly in the model block, but it could be safely removed from the program without changing sampling behavior. This is because $\log \mathsf{uniform}(\beta_{\mathsf{unif}} \mid -\pi/2, \pi/2) = -\log \pi$ is a constant and Stan only needs the total log probability up to an additive constant. Stan will spend some time checking that that `beta_unif` is between `-pi()/2` and `pi()/2`, but this condition is guaranteed by the constraints in the declaration of `beta_unif`.

Reparameterizing a Student-t Distribution

One thing that sometimes works when you're having trouble with the heavy-tailedness of Student-t distributions is to use the gamma-mixture representation, which says that you can generate a Student-t distributed variable β,

$$\beta \sim \mathsf{Student\text{-}t}(\nu, 0, 1),$$

by first generating a gamma-distributed precision (inverse variance) τ according to

$$\tau \sim \mathsf{Gamma}(\nu/2, \nu/2),$$

and then generating β from the normal distribution,

$$\beta \sim \mathsf{normal}\left(0, \tau^{-\frac{1}{2}}\right).$$

Because τ is precision, $\tau^{-\frac{1}{2}}$ is the scale (standard deviation), which is the parameterization used by Stan.

The marginal distribution of β when you integrate out τ is $\mathsf{Student\text{-}t}(\nu, 0, 1)$, i.e.,

$$\mathsf{Student\text{-}t}(\beta \mid \nu, 0, 1) = \int_0^\infty \mathsf{normal}\left(\beta \middle| 0, \tau^{-0.5}\right) \times \mathsf{Gamma}\left(\tau | \nu/2, \nu/2\right) \, \mathsf{d}\tau.$$

To go one step further, instead of defining a β drawn from a normal with precision τ, define α to be drawn from a unit normal,

$$\alpha \sim \mathsf{normal}(0, 1)$$

and rescale by defining

$$\beta = \alpha \, \tau^{-\frac{1}{2}}.$$

Now suppose $\mu = \beta x$ is the product of β with a regression predictor x. Then the reparameterization $\mu = \alpha\tau^{-\frac{1}{2}}x$ has the same distribution, but in the original, direct parameterization, β has (potentially) heavy tails, whereas in the second, neither τ nor α have heavy tails.

To translate into Stan notation, this reparameterization replaces

```
parameters {
  real<lower=0> nu;
  real beta;
  ...
model {
  beta ~ student_t(nu, 0, 1);
  ...
```

with

```
parameters {
  real<lower=0> nu;
  real<lower=0> tau;
  real alpha;
  ...
transformed parameters {
  real beta;
  beta = alpha / sqrt(tau);
  ...
model {
  real half_nu;
  half_nu = 0.5 * nu;
  tau ~ gamma(half_nu, half_nu);
  alpha ~ std_normal();
  ...
```

Although set to `0` here, in most cases, the lower bound for the degrees of freedom parameter `nu` can be set to `1` or higher; when `nu` is 1, the result is a Cauchy distribution with fat tails and as `nu` approaches infinity, the Student-t distribution approaches a normal distribution. Thus the parameter `nu` characterizes the heaviness of the tails of the model.

Hierarchical Models and the Non-Centered Parameterization

Unfortunately, the usual situation in applied Bayesian modeling involves complex geometries and interactions that are not known analytically. Nevertheless, reparameterization can still be effective for separating parameters.

Centered parameterization

For example, a vectorized hierarchical model might draw a vector of coefficients β with definitions as follows. The so-called centered parameterization is as follows.

```
parameters {
  real mu_beta;
  real<lower = 0> sigma_beta;
  vector[K] beta;
  ...
model {
  beta ~ normal(mu_beta, sigma_beta);
  ...
```

Although not shown, a full model will have priors on both `mu_beta` and `sigma_beta` along with data modeled based on these coefficients. For instance, a standard binary logistic regression with data matrix `x` and binary outcome vector `y` would include a likelihood statement such as form `y ~ bernoulli_logit(x * beta)`, leading to an analytically intractable posterior.

A hierarchical model such as the above will suffer from the same kind of inefficiencies as Neal's funnel, because the values of `beta`, `mu_beta` and `sigma_beta` are highly correlated in the posterior. The extremity of the correlation depends on the amount of data, with Neal's funnel being the extreme with no data. In these cases, the non-centered parameterization, discussed in the next section, is preferable; when there is a lot of data, the centered parameterization is more efficient. See Betancourt and Girolami (2013) for more information on the effects of centering in hierarchical models fit with Hamiltonian Monte Carlo.

Non-Centered Parameterization

Sometimes the group-level effects do not constrain the hierarchical distribution tightly. Examples arise when there are not many groups, or when the inter-group variation is high. In such cases, hierarchical models can be made much more efficient by shifting the data's correlation with the parameters to the hyperparameters. Similar

to the funnel example, this will be much more efficient in terms of effective sample size when there is not much data (see Betancourt and Girolami (2013)), and in more extreme cases will be necessary to achieve convergence.

```
parameters {
  vector[K] beta_raw;
  ...
transformed parameters {
  vector[K] beta;
  // implies: beta ~ normal(mu_beta, sigma_beta)
  beta = mu_beta + sigma_beta * beta_raw;
model {
  beta_raw ~ std_normal();
  ...
```

Any priors defined for `mu_beta` and `sigma_beta` remain as defined in the original model.

Reparameterization of hierarchical models is not limited to the normal distribution, although the normal distribution is the best candidate for doing so. In general, any distribution of parameters in the location-scale family is a good candidate for reparameterization. Let $\beta = l + s\alpha$ where l is a location parameter and s is a scale parameter. The parameter l need not be the mean, s need not be the standard deviation, and neither the mean nor the standard deviation need to exist. If α and β are from the same distributional family but α has location zero and unit scale, while β has location l and scale s, then that distribution is a location-scale distribution. Thus, if α were a parameter and β were a transformed parameter, then a prior distribution from the location-scale family on α with location zero and unit scale implies a prior distribution on β with location l and scale s. Doing so would reduce the dependence between α, l, and s.

There are several univariate distributions in the location-scale family, such as the Student t distribution, including its special cases of the Cauchy distribution (with one degree of freedom) and the normal distribution (with infinite degrees of freedom). As shown above, if α is distributed standard normal, then β is distributed normal with mean $\mu = l$ and standard deviation $\sigma = s$. The logistic, the double exponential, the generalized extreme value distributions, and the stable distribution are also in the location-scale family.

Also, if z is distributed standard normal, then z^2 is distributed chi-squared with one degree of freedom. By summing the squares of K independent standard normal variates, one can obtain a single variate that is distributed chi-squared with K degrees

of freedom. However, for large K, the computational gains of this reparameterization may be overwhelmed by the computational cost of specifying K primitive parameters just to obtain one transformed parameter to use in a model.

Multivariate Reparameterizations

The benefits of reparameterization are not limited to univariate distributions. A parameter with a multivariate normal prior distribution is also an excellent candidate for reparameterization. Suppose you intend the prior for β to be multivariate normal with mean vector μ and covariance matrix Σ. Such a belief is reflected by the following code.

```
data {
  int<lower=2> K;
  vector[K] mu;
  cov_matrix[K] Sigma;
  ...
parameters {
  vector[K] beta;
  ...
model {
  beta ~ multi_normal(mu, Sigma);
  ...
```

In this case `mu` and `Sigma` are fixed data, but they could be unknown parameters, in which case their priors would be unaffected by a reparameterization of `beta`.

If α has the same dimensions as β but the elements of α are independently and identically distributed standard normal such that $\beta = \mu + L\alpha$, where $LL^\top = \Sigma$, then β is distributed multivariate normal with mean vector μ and covariance matrix Σ. One choice for L is the Cholesky factor of Σ. Thus, the model above could be reparameterized as follows.

```
data {
  int<lower=2> K;
  vector[K] mu;
  cov_matrix[K] Sigma;
  ...
transformed data {
  matrix[K, K] L;
  L = cholesky_decompose(Sigma);
}
```

```
parameters {
  vector[K] alpha;
  ...
transformed parameters {
  vector[K] beta;
  beta = mu + L * alpha;
}
model {
  alpha ~ std_normal();
  // implies: beta ~ multi_normal(mu, Sigma)
  ...
```

This reparameterization is more efficient for two reasons. First, it reduces dependence among the elements of `alpha` and second, it avoids the need to invert `Sigma` every time `multi_normal` is evaluated.

The Cholesky factor is also useful when a covariance matrix is decomposed into a correlation matrix that is multiplied from both sides by a diagonal matrix of standard deviations, where either the standard deviations or the correlations are unknown parameters. The Cholesky factor of the covariance matrix is equal to the product of a diagonal matrix of standard deviations and the Cholesky factor of the correlation matrix. Furthermore, the product of a diagonal matrix of standard deviations and a vector is equal to the elementwise product between the standard deviations and that vector. Thus, if for example the correlation matrix `Tau` were fixed data but the vector of standard deviations `sigma` were unknown parameters, then a reparameterization of `beta` in terms of `alpha` could be implemented as follows.

```
data {
  int<lower=2> K;
  vector[K] mu;
  corr_matrix[K] Tau;
  ...
transformed data {
  matrix[K, K] L;
  L = cholesky_decompose(Tau);
}
parameters {
  vector[K] alpha;
  vector<lower=0>[K] sigma;
  ...
transformed parameters {
```

```
  vector[K] beta;
  // This equals mu + diag_matrix(sigma) * L * alpha;
  beta = mu + sigma .* (L * alpha);
}
model {
  sigma ~ cauchy(0, 5);
  alpha ~ std_normal();
  // implies: beta ~ multi_normal(mu,
  //  diag_matrix(sigma) * L * L' * diag_matrix(sigma)))
  ...
```

This reparameterization of a multivariate normal distribution in terms of standard normal variates can be extended to other multivariate distributions that can be conceptualized as contaminations of the multivariate normal, such as the multivariate Student t and the skew multivariate normal distribution.

A Wishart distribution can also be reparameterized in terms of standard normal variates and chi-squared variates. Let L be the Cholesky factor of a $K \times K$ positive definite scale matrix S and let ν be the degrees of freedom. If

$$A = \begin{pmatrix} \sqrt{c_1} & 0 & \cdots & 0 \\ z_{21} & \sqrt{c_2} & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ z_{K1} & \cdots & z_{K(K-1)} & \sqrt{c_K} \end{pmatrix},$$

where each c_i is distributed chi-squared with $\nu - i + 1$ degrees of freedom and each z_{ij} is distributed standard normal, then $W = LAA^\top L^\top$ is distributed Wishart with scale matrix $S = LL^\top$ and degrees of freedom ν. Such a reparameterization can be implemented by the following Stan code:

```
data {
  int<lower=1> N;
  int<lower=1> K;
  int<lower=K+2> nu
  matrix[K, K] L; // Cholesky factor of scale matrix
  vector[K] mu;
  matrix[N, K] y;
  ...
parameters {
  vector<lower=0>[K] c;
  vector[0.5 * K * (K - 1)] z;
```

```
  ...
model {
  matrix[K, K] A;
  int count = 1;
  for (j in 1:(K-1)) {
    for (i in (j+1):K) {
      A[i, j] = z[count];
      count += 1;
    }
    for (i in 1:(j - 1)) {
      A[i, j] = 0.0;
    }
    A[j, j] = sqrt(c[j]);
  }
  for (i in 1:(K-1))
    A[i, K] = 0;
  A[K, K] = sqrt(c[K]);

  for (i in 1:K)
    c[i] ~ chi_square(nu - i + 1);

  z ~ std_normal();
  // implies: L * A * A' * L' ~ wishart(nu, L * L')
  y ~ multi_normal_cholesky(mu, L * A);
  ...
```

This reparameterization is more efficient for three reasons. First, it reduces dependence among the elements of `z` and second, it avoids the need to invert the covariance matrix, W every time `wishart` is evaluated. Third, if W is to be used with a multivariate normal distribution, you can pass LA to the more efficient `multi_normal_cholesky` function, rather than passing W to `multi_normal`.

If W is distributed Wishart with scale matrix S and degrees of freedom ν, then W^{-1} is distributed inverse Wishart with inverse scale matrix S^{-1} and degrees of freedom ν. Thus, the previous result can be used to reparameterize the inverse Wishart distribution. Since $W = LAA^\top L^\top$, $W^{-1} = L^{\top^{-1}} A^{\top^{-1}} A^{-1} L^{-1}$, where all four inverses exist, but $L^{-1^\top} = L^{\top^{-1}}$ and $A^{-1^\top} = A^{\top^{-1}}$. We can slightly modify the above Stan code for this case:

```
data {
  int<lower=1> K;
```

```
  int<lower=K+2> nu
  matrix[K, K] L; // Cholesky factor of scale matrix
  ...
transformed data {
  matrix[K, K] eye;
  matrix[K, K] L_inv;
  for (j in 1:K) {
    for (i in 1:K) {
      eye[i, j] = 0.0;
    }
    eye[j, j] = 1.0;
  }
  L_inv = mdivide_left_tri_low(L, eye);
}
parameters {
  vector<lower=0>[K] c;
  vector[0.5 * K * (K - 1)] z;
  ...
model {
  matrix[K, K] A;
  matrix[K, K] A_inv_L_inv;
  int count;
  count = 1;
  for (j in 1:(K-1)) {
    for (i in (j+1):K) {
      A[i, j] = z[count];
      count += 1;
    }
    for (i in 1:(j - 1)) {
      A[i, j] = 0.0;
    }
    A[j, j] = sqrt(c[j]);
  }
  for (i in 1:(K-1))
    A[i, K] = 0;
  A[K, K] = sqrt(c[K]);

  A_inv_L_inv = mdivide_left_tri_low(A, L_inv);
  for (i in 1:K)
```

```
    c[i] ~ chi_square(nu - i + 1);

  z ~ std_normal(); // implies: crossprod(A_inv_L_inv) ~
  // inv_wishart(nu, L_inv' * L_inv)
  ...
```

Another candidate for reparameterization is the Dirichlet distribution with all K shape parameters equal. Zyczkowski and Sommers (2001) shows that if θ_i is equal to the sum of β independent squared standard normal variates and $\rho_i = \frac{\theta_i}{\sum \theta_i}$, then the K-vector ρ is distributed Dirichlet with all shape parameters equal to $\frac{\beta}{2}$. In particular, if $\beta = 2$, then ρ is uniformly distributed on the unit simplex. Thus, we can make ρ be a transformed parameter to reduce dependence, as in:

```
data {
  int<lower=1> beta;
  ...
parameters {
  vector[beta] z[K];
  ...
transformed parameters {
  simplex[K] rho;
  for (k in 1:K)
    rho[k] = dot_self(z[k]); // sum-of-squares
  rho = rho / sum(rho);
}
model {
  for (k in 1:K)
    z[k] ~ std_normal();
  // implies: rho ~ dirichlet(0.5 * beta * ones)
  ...
```

22.8. Vectorization

Gradient Bottleneck

Stan spends the vast majority of its time computing the gradient of the log probability function, making gradients the obvious target for optimization. Stan's gradient calculations with algorithmic differentiation require a template expression to be allocated and constructed for each subexpression of a Stan program involving param-

eters or transformed parameters.[3] This section defines optimization strategies based on vectorizing these subexpressions to reduce the work done during algorithmic differentiation.

Vectorizing Summations

Because of the gradient bottleneck described in the previous section, it is more efficient to collect a sequence of summands into a vector or array and then apply the `sum()` operation than it is to continually increment a variable by assignment and addition. For example, consider the following code snippet, where `foo()` is some operation that depends on `n`.

```
for (n in 1:N)
  total += foo(n,...);
```

This code has to create intermediate representations for each of the `N` summands.

A faster alternative is to copy the values into a vector, then apply the `sum()` operator, as in the following refactoring.

```
{
  vector[N] summands;
  for (n in 1:N)
    summands[n] = foo(n,...);
  total = sum(summands);
}
```

Syntactically, the replacement is a statement block delineated by curly brackets (`{`, `}`), starting with the definition of the local variable `summands`.

Even though it involves extra work to allocate the `summands` vector and copy `N` values into it, the savings in differentiation more than make up for it. Perhaps surprisingly, it will also use substantially less memory overall than incrementing `total` within the loop.

Vectorization through Matrix Operations

The following program directly encodes a linear regression with fixed unit noise using a two-dimensional array `x` of predictors, an array `y` of outcomes, and an array `beta` of regression coefficients.

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

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

The following model computes the same log probability function as the previous model, even supporting the same input files for data and initialization.

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

Although it produces equivalent results, the dot product should not be replaced with a transpose and multiply, as in

```
y[n] ~ normal(x[n]' * beta, 1);
```

The relative inefficiency of the transpose and multiply approach is that the transposition operator allocates a new vector into which the result of the transposition is

copied. This consumes both time and memory.[4]

The inefficiency of transposition could itself be mitigated by reordering the product and pulling the transposition out of the loop, as follows.

```
...
transformed parameters {
  row_vector[K] beta_t;
  beta_t = beta';
}
model {
  for (n in 1:N)
    y[n] ~ normal(beta_t * x[n], 1);
}
```

The problem with transposition could be completely solved by directly encoding the `x` as a row vector, as in the following example.

```
data {
  ...
  row_vector[K] x[N];
  ...
}
parameters {
  vector[K] beta;
}
model {
  for (n in 1:N)
    y[n] ~ normal(x[n] * beta, 1);
}
```

Declaring the data as a matrix and then computing all the predictors at once using matrix multiplication is more efficient still, as in the example discussed in the next section.

Having said all this, the most efficient way to code this model is with direct matrix multiplication, as in

```
data {
```

[4]Future versions of Stan may remove this inefficiency by more fully exploiting expression templates inside the Eigen C++ matrix library. This will require enhancing Eigen to deal with mixed-type arguments, such as the type `double` used for constants and the algorithmic differentiation type `stan::math::var` used for variables.

```
  matrix[N, K] x;
  vector[N] y;
}
parameters {
  vector[K] beta;
}
model {
  y ~ normal(x * beta, 1);
}
```

In general, encapsulated single operations that do the work of loops will be more efficient in their encapsulated forms. Rather than performing a sequence of row-vector/vector multiplications, it is better to encapsulate it as a single matrix/vector multiplication.

Vectorized Probability Functions

The final and most efficient version replaces the loops and transformed parameters by using the vectorized form of the normal probability function, as in the following example.

```
data {
  int<lower=1> K;
  int<lower=1> N;
  matrix[N, K] x;
  vector[N] y;
}
parameters {
  vector[K] beta;
}
model {
  y ~ normal(x * beta, 1);
}
```

The variables are all declared as either matrix or vector types. The result of the matrix-vector multiplication `x * beta` in the model block is a vector of the same length as `y`.

The probability function documentation in the function reference manual indicates which of Stan's probability functions support vectorization; see the function reference manual for full details. Vectorized probability functions accept either vector or scalar inputs for all arguments, with the only restriction being that all vector arguments

are the same dimensionality. In the example above, `y` is a vector of size `N`, `x * beta` is a vector of size `N`, and `1` is a scalar.

Reshaping Data for Vectorization

Sometimes data does not arrive in a shape that is ideal for vectorization, but can be put into such shape with some munging (either inside Stan's transformed data block or outside).

John Hall provided a simple example on the Stan users group. Simplifying notation a bit, the original model had a sampling statement in a loop, as follows.

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

The brute force vectorization would build up a mean vector and then vectorize all at once.

```
{
  vector[N] mu_ii;
  for (n in 1:N)
    mu_ii[n] = mu[ii[n]];
  y ~ normal(mu_ii, sigma);
}
```

If there aren't many levels (values `ii[n]` can take), then it behooves us to reorganize the data by group in a case like this. Rather than having a single observation vector `y`, there are K of them. And because Stan doesn't support ragged arrays, it means K declarations. For instance, with 5 levels, we have

```
y_1 ~ normal(mu[1], sigma);
...
y_5 ~ normal(mu[5], sigma);
```

This way, both the `mu` and `sigma` parameters are shared. Which way works out to be more efficient will depend on the shape of the data; if the sizes are small, the simple vectorization may be faster, but for moderate to large sized groups, the full expansion should be faster.

22.9. Exploiting Sufficient Statistics

In some cases, models can be recoded to exploit sufficient statistics in estimation. This can lead to large efficiency gains compared to an expanded model. For example, consider the following Bernoulli sampling model.

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

In this model, the sum of positive outcomes in `y` is a sufficient statistic for the chance of success `theta`. The model may be recoded using the binomial distribution as follows.

```
theta ~ beta(alpha, beta);
sum(y) ~ binomial(N, theta);
```

Because truth is represented as one and falsehood as zero, the sum `sum(y)` of a binary vector `y` is equal to the number of positive outcomes out of a total of `N` trials.

This can be generalized to other discrete cases (one wouldn't expect continuous observations to be duplicated if they are random). Suppose there are only K possible discrete outcomes, $z_1, \ldots, z_K$, but there are N observations, where N is much larger than K. If f_k is the frequency of outcome z_k, then the entire likelihood with distribution `foo` can be coded as follows.

```
for (k in 1:K)
  target += f[k] * foo_lpmf(z[k] | ...);
```

where the ellipses are the parameters of the log probability mass function for distribution `foo` (there's no distribution called "foo"; this is just a placeholder for any discrete distribution name).

The resulting program looks like a "weighted" regression, but here the weights `f[k]` are counts and thus sufficient statistics for the PMF and simply amount to an alternative, more efficient coding of the same likelihood. For efficiency, the frequencies `f[k]` should be counted once in the transformed data block and stored.

22.10. Aggregating Common Subexpressions

If an expression is calculated once, the value should be saved and reused wherever possible. That is, rather than using `exp(theta)` in multiple places, declare a local variable to store its value and reuse the local variable.

Another case that may not be so obvious is with two multilevel parameters, say `a[ii[n]] + b[jj[n]]`. If `a` and `b` are small (i.e., do not have many levels), then a table `a_b` of their sums can be created, with

```
matrix[size(a), size(b)] a_b;
for (i in 1:size(a))
  for (j in 1:size(b))
    a_b[i, j] = a[i] + b[j];
```

Then the sum can be replaced with `a_b[ii[n], jj[n]]`.

22.11. Exploiting Conjugacy

Continuing the model from the previous section, the conjugacy of the beta prior and binomial sampling distribution allow the model to be further optimized to the following equivalent form.

```
theta ~ beta(alpha + sum(y), beta + N - sum(y));
```

To make the model even more efficient, a transformed data variable defined to be `sum(y)` could be used in the place of `sum(y)`.

22.12. Standardizing Predictors and Outputs

Stan programs will run faster if the input is standardized to have a zero sample mean and unit sample variance. This section illustrates the principle with a simple linear regression.

Suppose that $y = (y_1, \ldots, y_N)$ is a sequence of N outcomes and $x = (x_1, \ldots, x_N)$ a parallel sequence of N predictors. A simple linear regression involving an intercept coefficient α and slope coefficient β can be expressed as

$$y_n = \alpha + \beta x_n + \epsilon_n,$$

where

$$\epsilon_n \sim \mathsf{normal}(0, \sigma).$$

If either vector x or y has very large or very small values or if the sample mean of the values is far away from 0 (on the scale of the values), then it can be more

efficient to standardize the outputs y_n and predictors x_n. The data are first centered by subtracting the sample mean, and then scaled by dividing by the sample deviation. Thus a data point u is standardized with respect to a vector y by the function $\mathtt{z}_y$, defined by

$$\mathtt{z}_y(u) = \frac{u - \bar{y}}{\mathtt{sd}(y)}$$

where the sample mean of y is

$$\bar{y} = \frac{1}{N} \sum_{n=1}^{N} y_n,$$

and the sample standard deviation of y is

$$\mathtt{sd}(y) = \left(\frac{1}{N} \sum_{n=1}^{N} (y_n - \bar{y})^2 \right)^{1/2}.$$

The inverse transform is defined by reversing the two normalization steps, first rescaling by the same deviation and relocating by the sample mean,

$$\mathtt{z}_y^{-1}(v) = \mathtt{sd}(y) v + \bar{y}.$$

To standardize a regression problem, the predictors and outcomes are standardized. This changes the scale of the variables, and hence changes the scale of the priors. Consider the following initial model.

```
data {
  int<lower=0> N;
  vector[N] y;
  vector[N] x;
}
parameters {
  real alpha;
  real beta;
  real<lower=0> sigma;
}
model {
  // priors
  alpha ~ normal(0, 10);
  beta ~ normal(0, 10);
  sigma ~ cauchy(0, 5);
```

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

The data block for the standardized model is identical. The standardized predictors and outputs are defined in the transformed data block.

```
data {
  int<lower=0> N;
  vector[N] y;
  vector[N] x;
}
transformed data {
  vector[N] x_std;
  vector[N] y_std;
  x_std = (x - mean(x)) / sd(x);
  y_std = (y - mean(y)) / sd(y);
}
parameters {
  real alpha_std;
  real beta_std;
  real<lower=0> sigma_std;
}
model {
  alpha_std ~ normal(0, 10);
  beta_std ~ normal(0, 10);
  sigma_std ~ cauchy(0, 5);
  for (n in 1:N)
    y_std[n] ~ normal(alpha_std + beta_std * x_std[n],
                      sigma_std);
}
```

The parameters are renamed to indicate that they aren't the "natural" parameters, but the model is otherwise identical. In particular, the fairly diffuse priors on the coefficients and error scale are the same. These could have been transformed as well, but here they are left as is, because the scales make sense as diffuse priors for standardized data; the priors could be made more informative. For instance, because the outputs y have been standardized, the error σ should not be greater than 1, because that's the scale of the noise for predictors $\alpha = \beta = 0$.

The original regression

$$y_n = \alpha + \beta x_n + \epsilon_n$$

has been transformed to a regression on the standardized variables,

$$\mathtt{z}_y(y_n) = \alpha' + \beta' \mathtt{z}_x(x_n) + \epsilon'_n.$$

The original parameters can be recovered with a little algebra,

$$\begin{aligned}
y_n &= \mathtt{z}_y^{-1}(\mathtt{z}_y(y_n)) \\
&= \mathtt{z}_y^{-1}\left(\alpha' + \beta' \mathtt{z}_x(x_n) + \epsilon'_n\right) \\
&= \mathtt{z}_y^{-1}\left(\alpha' + \beta'\left(\frac{x_n - \bar{x}}{\mathtt{sd}(x)}\right) + \epsilon'_n\right) \\
&= \mathtt{sd}(y)\left(\alpha' + \beta'\left(\frac{x_n - \bar{x}}{\mathtt{sd}(x)}\right) + \epsilon'_n\right) + \bar{y} \\
&= \left(\mathtt{sd}(y)\left(\alpha' - \beta'\frac{\bar{x}}{\mathtt{sd}(x)}\right) + \bar{y}\right) + \left(\beta'\frac{\mathtt{sd}(y)}{\mathtt{sd}(x)}\right) x_n + \mathtt{sd}(y)\epsilon'_n,
\end{aligned}$$

from which the original scale parameter values can be read off,

$$\alpha = \mathtt{sd}(y)\left(\alpha' - \beta'\frac{\bar{x}}{\mathtt{sd}(x)}\right) + \bar{y}; \qquad \beta = \beta'\frac{\mathtt{sd}(y)}{\mathtt{sd}(x)}; \qquad \sigma = \mathtt{sd}(y)\sigma'.$$

These recovered parameter values on the original scales can be calculated within Stan using a generated quantities block following the model block,

```
generated quantities {
  real alpha;
  real beta;
  real<lower=0> sigma;
  alpha = sd(y) * (alpha_std - beta_std * mean(x) / sd(x))
          + mean(y);
  beta = beta_std * sd(y) / sd(x);
  sigma = sd(y) * sigma_std;
}
```

It is inefficient to compute all of the means and standard deviations every iteration; for more efficiency, these can be calculated once and stored as transformed data. Furthermore, the model sampling statement can be easily vectorized, for instance, in the transformed model, to

```
y_std ~ normal(alpha_std + beta_std * x_std, sigma_std);
```

Standard Normal Distribution

For many applications on the standard scale, normal distributions with location zero and scale one will be used. In these cases, it is more efficient to use

```
y ~ std_normal();
```

than to use

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

because the subtraction of the location and division by the scale cancel, as does subtracting the log of the scale.

22.13. Using Map-Reduce

The map-reduce operation, even without multi-core MPI support, can be used to make programs more scalable and also more efficient. See the map-reduce chapter for more information on implementing map-reduce operations.

Map-reduce allows greater scalability because only the Jacobian of the mapped function for each shard is stored. The Jacobian consists of all of the derivatives of the outputs with respect to the parameters. During execution, the derivatives of the shard are evaluated using nested automatic differentiation. As often happens with modern CPUs, reduced memory overhead leads to increased memory locality and faster execution. The Jacobians are all computed with local memory and their outputs stored contiguously in memory.

23. Map-Reduce

Map-reduce allows large calculations (e.g., log likelihoods) to be broken into components which may be calculated modularly (e.g., data blocks) and combined (e.g., by summation and incrementing the target log density).

23.1. Overview of Map-Reduce

A *map function* is a higher-order function that applies an argument function to every member of some collection, returning a collection of the results. For example, mapping the square function, $f(x) = x^2$, over the vector $[3, 5, 10]$ produces the vector $[9, 25, 100]$. In other words, map applies the square function elementwise.

The output of mapping a sequence is often fed into a reduction. A *reduction function* takes an arbitrarily long sequence of inputs and returns a single output. Examples of reduction functions are summation (with the return being a single value) or sorting (with the return being a sorted sequence). The combination of mapping and reducing is so common it has its own name, *map-reduce*.

23.2. Map Function

In order to generalize the form of functions and results that are possible and accommodate both parameters (which need derivatives) and data values (which don't), Stan's map function operates on more than just a sequence of inputs.

Map Function Signature

Stan's map function has the following signature

```
vector map_rect((vector, vector, real[], int[]):vector f,
                vector phi, vector[] thetas,
                data real[ , ] x_rs, data int[ , ] x_is);
```

The arrays `thetas` of parameters, `x_rs` of real data, and `x_is` of integer data have the suffix "`s`" to indicate they are arrays. These arrays must all be the same size, as they will be mapped in parallel by the function `f`. The value of `phi` is reused in each mapped operation.

The `_rect` suffix in the name arises because the data structures it takes as arguments are rectangular. In order to deal with ragged inputs, ragged inputs must be padded

out to rectangular form.

The last two arguments are two dimensional arrays of real and integer data values. These argument types are marked with the `data` qualifier to indicate that they must only contain variables originating in the data or transformed data blocks. This will allow such data to be pinned to a processor on which it is being processed to reduce communication overhead.

The notation `(vector, vector, real[], int[]):vector` indicates that the function argument `f` must have the following signature.

```
vector f(vector phi, vector theta,
         data real[] x_r, data int[] x_i);
```

Although `f` will often return a vector of size one, the built-in flexibility allows general multivariate functions to be mapped, even raggedly.

Map Function Semantics

Stan's map function applies the function `f` to the shared parameters along with one element each of the job parameters, real data, and integer data arrays. Each of the arguments `theta`, `x_r`, and `x_i` must be arrays of the same size. If the arrays are all size `N`, the result is defined as follows.

```
map_rect(f, phi, thetas, xs, ns)
= f(phi, thetas[1], xs[1], ns[1]) . f(phi, thetas[2], xs[2], ns[2])
  . ... . f(phi, thetas[N], xs[N], ns[N])
```

The dot operators in the notation above are meant to indicate concatenation (implemented as `append_row` in Stan). The output of each application of `f` is a vector, and the sequence of `N` vectors is concatenated together to return a single vector.

23.3. Example: Mapping Logistic Regression

An example should help to clarify both the syntax and semantics of the mapping operation and how it may be combined with reductions built into Stan to provide a map-reduce implementation.

Unmapped Logistic Regression

Consider the following simple logistic regression model, which is coded unconventionally to accomodate direct translation to a mapped implementation.

```
data {
```

```
  int y[12];
  real x[12];
}
parameters {
  vector[2] beta;
}
model {
  beta ~ std_normal();
  y ~ bernoulli_logit(beta[1] + beta[2] * to_vector(x));
}
```

The program is unusual in that it (a) hardcodes the data size, which is not required by the map function but is just used here for simplicity, (b) represents the predictors as a real array even though it needs to be used as a vector, and (c) represents the regression coefficients (intercept and slope) as a vector even though they're used individually. The `bernoulli_logit` distribution is used because the argument is on the logit scale—it implicitly applies the inverse logit function to map the argument to a probability.

Mapped Logistic Regression

The unmapped logistic regression model described in the previous subsection may be implemented using Stan's rectangular mapping functionality as follows.

```
functions {
  vector lr(vector beta, vector theta, real[] x, int[] y) {
    real lp = bernoulli_logit_lpmf(y | beta[1]
                                     + to_vector(x) * beta[2]);
    return [lp]';
  }
}
data {
  int y[12];
  real x[12];
}
transformed data {
  // K = 3 shards
  int ys[3, 4] = { y[1:4], y[5:8], y[9:12] };
  real xs[3, 4] = { x[1:4], x[5:8], x[9:12] };
  vector[0] theta[3];
}
```

```
parameters {
  vector[2] beta;
}
model {
  beta ~ std_normal();
  target += sum(map_rect(lr, beta, theta, xs, ys));
}
```

The first piece of the code is the actual function to compute the logistic regression. The argument `beta` will contain the regression coefficients (intercept and slope), as before. The second argument `theta` of job-specific parameters is not used, but nevertheless must be present. The modeled data `y` is passed as an array of integers and the predictors `x` as an array of real values. The function body then computes the log probability mass of `y` and assigns it to the local variable `lp`. This variable is then used in `[lp]'` to construct a row vector and then transpose it to a vector to return.

The data are taken in as before. There is an additional transformed data block that breaks the data up into three shards.[1]

The value `3` is also hard coded; a more practical program would allow the number of shards to be controlled. There are three parallel arrays defined here, each of size three, corresponding to the number of shards. The array `ys` contains the modeled data variables; each element of the array `ys` is an array of size four. The second array `xs` is for the predictors, and each element of it is also of size four. These contained arrays are the same size because the predictors `x` stand in a one-to-one relationship with the modeled data `y`. The final array `theta` is also of size three; its elements are empty vectors, because there are no shard-specific parameters.

The parameters and the prior are as before. The likelihood is now coded using map-reduce. The function `lr` to compute the log probability mass is mapped over the data `xs` and `ys`, which contain the original predictors and outcomes broken into shards. The parameters `beta` are in the first argument because they are shared across shards. There are no shard-specific parameters, so the array of job-specific parameters `theta` contains only empty vectors.

[1] The term "shard" is borrowed from databases, where it refers to a slice of the rows of a database. That is exactly what it is here if we think of rows of a dataframe. Stan's shards are more general in that they need not correspond to rows of a dataframe.

23.4. Example: Hierarchical Logistic Regression

Consider a hierarchical model of American presidential voting behavior based on state of residence.[2]

Each of the fifty states $k \in \{1, \ldots, 50\}$ will have its own slope β_k and intercept α_k to model the log odds of voting for the Republican candidate as a function of income. Suppose there are N voters and with voter $n \in 1{:}N$ being in state $s[n]$ with income x_n. The likelihood for the vote $y_n \in \{0, 1\}$ is

$$y_n \sim \mathsf{Bernoulli}\Big(\text{logit}^{-1} \left(\alpha_{s[n]} + \beta_{s[n]} \, x_n \right) \Big).$$

The slopes and intercepts get hierarchical priors,

$$\alpha_k \sim \mathsf{normal}(\mu_\alpha, \sigma_\alpha)$$
$$\beta_k \sim \mathsf{normal}(\mu_\beta, \sigma_\beta)$$

Unmapped Implementation

This model can be coded up in Stan directly as follows.

```
data {
  int<lower = 0> K;
  int<lower = 0> N;
  int<lower = 1, upper = K> kk[N];
  vector[N] x;
  int<lower = 0, upper = 1> y[N];
}
parameters {
  matrix[K,2] beta;
  vector[2] mu;
  vector<lower=0>[2] sigma;
}
model {
  mu ~ normal(0, 2);
  sigma ~ normal(0, 2);
  for (i in 1:2)
    beta[ , i] ~ normal(mu[i], sigma[i]);
```

[2]This example is a simplified form of the model described in (Gelman and Hill 2007, Section 14.2)

```
  y ~ bernoulli_logit(beta[kk, 1] + beta[kk, 2] .* x);
}
```

For this model the vector of predictors `x` is coded as a vector, corresponding to how it is used in the likelihood. The priors for `mu` and `sigma` are vectorized. The priors on the two components of `beta` (intercept and slope, respectively) are stored in a $K \times 2$ matrix.

The likelihood is also vectorized using multi-indexing with index `kk` for the states and elementwise multiplication (`.*`) for the income `x`. The vectorized likelihood works out to the same thing as the following less efficient looped form.

```
for (n in 1:N)
  y[n] ~ bernoulli_logit(beta[kk[n], 1] + beta[kk[n], 2] * x[n]);
```

Mapped Implementation

The mapped version of the model will map over the states `K`. This means the group-level parameters, real data, and integer-data must be arrays of the same size.

The mapped implementation requires a function to be mapped. The following function evaluates both the likelihood for the data observed for a group as well as the prior for the group-specific parameters (the name `bl_glm` derives from the fact that it's a generalized linear model with a Bernoulli likelihood and logistic link function).

```
functions {
 vector bl_glm(vector mu_sigma, vector beta,
               real[] x, int[] y) {
   vector[2] mu = mu_sigma[1:2];
   vector[2] sigma = mu_sigma[3:4];
   real lp = normal_lpdf(beta | mu, sigma);
   real ll = bernoulli_logit_lpmf(y | beta[1] + beta[2] * to_vector(x));
   return [lp + ll]';
 }
}
```

The shared parameter `mu_sigma` contains the locations (`mu_sigma[1:2]`) and scales (`mu_sigma[3:4]`) of the priors, which are extracted in the first two lines of the program. The variable `lp` is assigned the log density of the prior on `beta`. The vector `beta` is of size two, as are the vectors `mu` and `sigma`, so everything lines up for the vectorization. Next, the variable `ll` is assigned to the log likelihood contribution for

the group. Here `beta[1]` is the intercept of the regression and `beta[2]` the slope. The predictor array `x` needs to be converted to a vector allow the multiplication.

The data block is identical to that of the previous program, but repeated here for convenience. A transformed data block computes the data structures needed for the mapping by organizing the data into arrays indexed by group.

```
data {
  int<lower = 0> K;
  int<lower = 0> N;
  int<lower = 1, upper = K> kk[N];
  vector[N] x;
  int<lower = 0, upper = 1> y[N];
}
transformed data {
  int<lower = 0> J = N / K;
  real x_r[K, J];
  int<lower = 0, upper = 1> x_i[K, J];
  {
    int pos = 1;
    for (k in 1:K) {
      int end = pos + J - 1;
      x_r[k] = to_array_1d(x[pos:end]);
      x_i[k] = to_array_1d(y[pos:end]);
      pos += J;
    }
  }
}
```

The integer `J` is set to the number of observations per group.[3]

The real data array `x_r` holds the predictors and the integer data array `x_i` holds the outcomes. The grouped data arrays are constructed by slicing the predictor vector `x` (and converting it to an array) and slicing the outcome array `y`.

Given the transformed data with groupings, the parameters are the same as the previous program. The model has the same priors for the hyperparameters `mu` and `sigma`, but moves the prior for `beta` and the likelihood to the mapped function.

```
parameters {
  vector[2] beta[K];
```

[3]This makes the strong assumption that each group has the same number of observations!

```
  vector[2] mu;
  vector<lower=0>[2] sigma;
}
model {
  mu ~ normal(0, 2);
  sigma ~ normal(0, 2);
  target += sum(map_rect(bl_glm, append_row(mu, sigma), beta, x_r, x_i));

}
```

The model as written here computes the priors for each group's parameters along with the likelihood contribution for the group. An alternative mapping would leave the prior in the model block and only map the likelihood. In a serial setting this shouldn't make much of a difference, but with parallelization, there is reduced communication (the prior's parameters need not be transmitted) and also reduced parallelization with the version that leaves the prior in the model block.

23.5. Ragged Inputs and Outputs

The previous examples included rectangular data structures and single outputs. Despite the name, this is not technically required by `map_rect`.

Ragged Inputs

If each group has a different number of observations, then the rectangular data structures for predictors and outcomes will need to be padded out to be rectangular. In addition, the size of the ragged structure will need to be passed as integer data. This holds for shards with varying numbers of parameters as well as varying numbers of data points.

Ragged Outputs

The output of each mapped function is concatenated in order of inputs to produce the output of `map_rect`. When every shard returns a singleton (size one) array, the result is the same size as the number of shards and is easy to deal with downstream. If functions return longer arrays, they can still be structured using the `to_matrix` function if they are rectangular.

If the outputs are of varying sizes, then there will have to be some way to convert it back to a usable form based on the input, because there is no way to directly return sizes or a ragged structure.

Appendices

These are the appendices for the book, gathered here as they are not part of the main exposition.

24. Stan Program Style Guide

This chapter describes the preferred style for laying out Stan models. These are not rules of the language, but simply recommendations for laying out programs in a text editor. Although these recommendations may seem arbitrary, they are similar to those of many teams for many programming languages. Like rules for typesetting text, the goal is to achieve readability without wasting white space either vertically or horizontally.

24.1. Choose a Consistent Style

The most important point of style is consistency. Consistent coding style makes it easier to read not only a single program, but multiple programs. So when departing from this style guide, the number one recommendation is to do so consistently.

24.2. Line Length

Line lengths should not exceed 80 characters.[1]

This is a typical recommendation for many programming language style guides because it makes it easier to lay out text edit windows side by side and to view the code on the web without wrapping, easier to view diffs from version control, etc. About the only thing that is sacrificed is laying out expressions on a single line.

24.3. File Extensions

The recommended file extension for Stan model files is `.stan`. For Stan data dump files, the recommended extension is `.R`, or more informatively, `.data.R`.

24.4. Variable Naming

The recommended variable naming is to follow C/C++ naming conventions, in which variables are lowercase, with the underscore character (`_`) used as a separator. Thus it is preferred to use `sigma_y`, rather than the run together `sigmay`, camel-case `sigmaY`, or capitalized camel-case `SigmaY`. Even matrix variables should be lowercased.

[1] Even 80 characters may be too many for rendering in print; for instance, in this manual, the number of code characters that fit on a line is about 65.

The exception to the lowercasing recommendation, which also follows the C/C++ conventions, is for size constants, for which the recommended form is a single uppercase letter. The reason for this is that it allows the loop variables to match. So loops over the indices of an $M \times N$ matrix a would look as follows.

```
for (m in 1:M)
  for (n in 1:N)
     a[m,n] = ...
```

24.5. Local Variable Scope

Declaring local variables in the block in which they are used aids in understanding programs because it cuts down on the amount of text scanning or memory required to reunite the declaration and definition.

The following Stan program corresponds to a direct translation of a BUGS model, which uses a different element of `mu` in each iteration.

```
model {
  real mu[N];
  for (n in 1:N) {
    mu[n] = alpha * x[n] + beta;
    y[n] ~ normal(mu[n],sigma);
  }
}
```

Because variables can be reused in Stan and because they should be declared locally for clarity, this model should be recoded as follows.

```
model {
  for (n in 1:N) {
    real mu;
    mu = alpha * x[n] + beta;
    y[n] ~ normal(mu,sigma);
  }
}
```

The local variable can be eliminated altogether, as follows.

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

There is unlikely to be any measurable efficiency difference between the last two implementations, but both should be a bit more efficient than the BUGS translation.

Scope of Compound Structures with Componentwise Assignment

In the case of local variables for compound structures, such as arrays, vectors, or matrices, if they are built up component by component rather than in large chunks, it can be more efficient to declare a local variable for the structure outside of the block in which it is used. This allows it to be allocated once and then reused.

```
model {
  vector[K] mu;
  for (n in 1:N) {
    for (k in 1:K)
      mu[k] = ...;
    y[n] ~ multi_normal(mu,Sigma);
}
```

In this case, the vector `mu` will be allocated outside of both loops, and used a total of `N` times.

24.6. Parentheses and Brackets

Optional Parentheses for Single-Statement Blocks

Single-statement blocks can be rendered in one of two ways. The fully explicit bracketed way is as follows.

```
for (n in 1:N) {
  y[n] ~ normal(mu,1);
}
```

The following statement without brackets has the same effect.

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

Single-statement blocks can also be written on a single line, as in the following example.

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

These can be much harder to read than the first example. Only use this style if the statement is simple, as in this example. Unless there are many similar cases, it's almost always clearer to put each sampling statement on its own line.

Conditional and looping statements may also be written without brackets.

The use of for loops without brackets can be dangerous. For instance, consider this program.

```
for (n in 1:N)
  z[n] ~ normal(nu,1);
  y[n] ~ normal(mu,1);
```

Because Stan ignores whitespace and the parser completes a statement as eagerly as possible (just as in C++), the previous program is equivalent to the following program.

```
for (n in 1:N) {
  z[n] ~ normal(nu,1);
}
y[n] ~ normal(mu,1);
```

Parentheses in Nested Operator Expressions

The preferred style for operators minimizes parentheses. This reduces clutter in code that can actually make it harder to read expressions. For example, the expression `a + b * c` is preferred to the equivalent `a + (b * c)` or `(a + (b * c))`. The operator precedences and associativities follow those of pretty much every programming language including Fortran, C++, R, and Python; full details are provided in the reference manual.

Similarly, comparison operators can usually be written with minimal bracketing, with the form `y[n] > 0 || x[n] != 0` preferred to the bracketed form `(y[n] > 0) || (x[n] != 0)`.

No Open Brackets on Own Line

Vertical space is valuable as it controls how much of a program you can see. The preferred Stan style is as shown in the previous section, not as follows.

```
for (n in 1:N)
{
  y[n] ~ normal(mu,1);
}
```

This also goes for parameters blocks, transformed data blocks, which should look as follows.

```
transformed parameters {
  real sigma;
  ...
}
```

24.7. Conditionals

Stan supports the full C++-style conditional syntax, allowing real or integer values to act as conditions, as follows.

```
real x;
...
if (x) {
   // executes if x not equal to 0
   ...
}
```

Explicit Comparisons of Non-Boolean Conditions

The preferred form is to write the condition out explicitly for integer or real values that are not produced as the result of a comparison or boolean operation, as follows.

```
if (x != 0) ...
```

24.8. Functions

Functions are laid out the same way as in languages such as Java and C++. For example,

```
real foo(real x, real y) {
  return sqrt(x * log(y));
}
```

The return type is flush left, the parentheses for the arguments are adjacent to the arguments and function name, and there is a space after the comma for arguments after the first. The open curly brace for the body is on the same line as the function name, following the layout of loops and conditionals. The body itself is indented; here we use two spaces. The close curly brace appears on its own line.

If function names or argument lists are long, they can be written as

```
matrix
function_to_do_some_hairy_algebra(matrix thingamabob,
                                  vector doohickey2) {
  ...body...
}
```

The function starts a new line, under the type. The arguments are aligned under each other.

Function documentation should follow the Javadoc and Doxygen styles. Here's an example repeated from the documenting functions section.

```
/**
 * Return a data matrix of specified size with rows
 * corresponding to items and the first column filled
 * with the value 1 to represent the intercept and the
 * remaining columns randomly filled with unit-normal draws.
 *
 * @param N Number of rows correspond to data items
 * @param K Number of predictors, counting the intercept, per
 *          item.
 * @return Simulated predictor matrix.
 */
matrix predictors_rng(int N, int K) {
  ...
```

The open comment is `/**`, asterisks are aligned below the first asterisk of the open comment, and the end comment `*/` is also aligned on the asterisk. The tags `@param` and `@return` are used to label function arguments (i.e., parameters) and return values.

24.9. White Space

Stan allows spaces between elements of a program. The white space characters allowed in Stan programs include the space (ASCII `0x20`), line feed (ASCII `0x0A`), carriage return (`0x0D`), and tab (`0x09`). Stan treats all whitespace characters interchangeably, with any sequence of whitespace characters being syntactically equivalent to a single space character. Nevertheless, effective use of whitespace is the key to good program layout.

Line Breaks Between Statements and Declarations

It is dispreferred to have multiple statements or declarations on the same line, as in the following example.

```
transformed parameters {
  real mu_centered;  real sigma;
  mu = (mu_raw - mean_mu_raw);    sigma = pow(tau,-2);
}
```

These should be broken into four separate lines.

No Tabs

Stan programs should not contain tab characters. They are legal and may be used anywhere other whitespace occurs. Using tabs to layout a program is highly unportable because the number of spaces represented by a single tab character varies depending on which program is doing the rendering and how it is configured.

Two-Character Indents

Stan has standardized on two space characters of indentation, which is the standard convention for C/C++ code. Another sensible choice is four spaces, which is the convention for Java and Python. Just be consistent.

Space Between `if`, `{` and Condition

Use a space after `if`s. For instance, use `if (x < y) ...`, not `if(x < y) ...`.

No Space For Function Calls

There is no space between a function name and the function it applies to. For instance, use `normal(0,1)`, not `normal (0,1)`.

Spaces Around Operators

There should be spaces around binary operators. For instance, use `y[1] = x`, not `y[1]=x`, use `(x + y) * z` not `(x+y)*z`.

Breaking Expressions across Lines

Sometimes expressions are too long to fit on a single line. In that case, the recommended form is to break *before* an operator,[2] aligning the operator to indicate scoping. For example, use the following form (though not the content; inverting matrices is almost always a bad idea).

```
target += (y - mu)' * inv(Sigma) * (y - mu);
```

Here, the multiplication operator (`*`) is aligned to clearly signal the multiplicands in the product.

For function arguments, break after a comma and line the next argument up underneath as follows.

```
y[n] ~ normal(alpha + beta * x + gamma * y,
              pow(tau,-0.5));
```

Optional Spaces after Commas

Optionally use spaces after commas in function arguments for clarity. For example, `normal(alpha * x[n] + beta,sigma)` can also be written as `normal(alpha * x[n] + beta, sigma)`.

Unix Newlines

Wherever possible, Stan programs should use a single line feed character to separate lines. All of the Stan developers (so far, at least) work on Unix-like operating systems and using a standard newline makes the programs easier for us to read and share.

Platform Specificity of Newlines

Newlines are signaled in Unix-like operating systems such as Linux and Mac OS X with a single line-feed (LF) character (ASCII code point `0x0A`). Newlines are signaled in Windows using two characters, a carriage return (CR) character (ASCII code point `0x0D`) followed by a line-feed (LF) character.

[2] This is the usual convention in both typesetting and other programming languages. Neither R nor BUGS allows breaks before an operator because they allow newlines to signal the end of an expression or statement.

25. Transitioning from BUGS

From the outside, Stan and BUGS[1] are similar—they use statistically-themed modeling languages (which are similar but with some differences; see below), they can be called from R, running some specified number of chains to some specified length, producing posterior simulations that can be assessed using standard convergence diagnostics. This is not a coincidence: in designing Stan: we wanted to keep many of the useful features of Bugs.

25.1. Some Differences in How BUGS and Stan Work

BUGS is interpreted, Stan is compiled

Stan is compiled in two steps, first a model is translated to templated C++ and then to a platform-specific executable. Stan, unlike BUGS, allows the user to directly program in C++, but we do not describe how to do this in this Stan manual (see the getting started with C++ section of https://mc-stan.org for more information on using Stan directly from C++).

BUGS performs MCMC updating one scalar parameter at a time, Stan uses HMC which moves in the entire space of all the parameters at each step

BUGS performs MCMC updating one scalar parameter at a time, (with some exceptions such as JAGS's implementation of regression and generalized linear models and some conjugate multivariate parameters), using conditional distributions (Gibbs sampling) where possible and otherwise using adaptive rejection sampling, slice sampling, and Metropolis jumping. BUGS figures out the dependence structure of the joint distribution as specified in its modeling language and uses this information to compute only what it needs at each step. Stan moves in the entire space of all the parameters using Hamiltonian Monte Carlo (more precisely, the no-U-turn sampler), thus avoiding some difficulties that occur with one-dimension-at-a-time sampling in high dimensions but at the cost of requiring the computation of the entire log density at each step.

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

Differences in tuning during warmup

BUGS tunes its adaptive jumping (if necessary) during its warmup phase (traditionally referred to as "burn-in"). Stan uses its warmup phase to tune the no-U-turn sampler (NUTS).

The Stan language is directly executable, the BUGS modeling language is not

The BUGS modeling language is not directly executable. Rather, BUGS parses its model to determine the posterior density and then decides on a sampling scheme. In contrast, the statements in a Stan model are directly executable: they translate exactly into C++ code that is used to compute the log posterior density (which in turn is used to compute the gradient).

Differences in statement order

In BUGS, statements are executed according to the directed graphical model so that variables are always defined when needed. A side effect of the direct execution of Stan's modeling language is that statements execute in the order in which they are written. For instance, the following Stan program, which sets `mu` before using it to sample `y`:

```
mu = a + b * x;
y ~ normal(mu, sigma);
```

translates to the following C++ code:

```
mu = a + b * x;
target += normal_lpdf(y | mu, sigma);
```

Contrast this with the following Stan program:

```
y ~ normal(mu, sigma);
mu = a + b * x;
```

This program is well formed, but is almost certainly a coding error, because it attempts to use `mu` before it is set. The direct translation to C++ code highlights the potential error of using `mu` in the first statement:

```
target += normal_lpdf(y | mu, sigma);
mu = a + b * x;
```

To trap these kinds of errors, variables are initialized to the special not-a-number (`NaN`) value. If `NaN` is passed to a log probability function, it will raise a domain

exception, which will in turn be reported by the sampler. The sampler will reject the sample out of hand as if it had zero probability.

Stan computes the gradient of the log density, BUGS computes the log density but not its gradient

Stan uses its own C++ algorithmic differentiation packages to compute the gradient of the log density (up to a proportion). Gradients are required during the Hamiltonian dynamics simulations within the leapfrog algorithm of the Hamiltonian Monte Carlo and NUTS samplers.

Both BUGS and Stan are semi-automatic

Both BUGS and Stan are semi-automatic in that they run by themselves with no outside tuning required. Nevertheless, the user needs to pick the number of chains and number of iterations per chain. We usually pick 4 chains and start with 10 iterations per chain (to make sure there are no major bugs and to approximately check the timing), then go to 100, 1000, or more iterations as necessary. Compared to Gibbs or Metropolis, Hamiltonian Monte Carlo can take longer per iteration (as it typically takes many "leapfrog steps" within each iteration), but the iterations typically have lower autocorrelation. So Stan might work fine with 1000 iterations in an example where BUGS would require 100,000 for good mixing. We recommend monitoring potential scale reduction statistics ($\hat{R}$) and the effective sample size to judge when to stop (stopping when $\hat{R}$ values do not counter-indicate convergence and when enough effective samples have been collected).

Licensing

WinBUGS is closed source. OpenBUGS and JAGS are both licensed under the Gnu Public License (GPL), otherwise known as copyleft due to the restrictions it places on derivative works. Stan is licensed under the much more liberal new BSD license.

Interfaces

Like WinBUGS, OpenBUGS and JAGS, Stan can be run directly from the command line or through common analytics platforms like R, Python, Julia, MATLAB, Mathematica, and the command line.

Platforms

Like OpenBUGS and JAGS, Stan can be run on Linux, Mac, and Windows platforms.

25.2. Some Differences in the Modeling Languages

The BUGS modeling language follows an R-like syntax in which line breaks are meaningful. Stan follows the rules of C, in which line breaks are equivalent to spaces, and each statement ends in a semicolon. For example:

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

and

```
for (i in 1:n) y[i] ~ normal(mu, sigma);
```

Or, equivalently (recall that a line break is just another form of whitespace),

```
for (i in 1:n)
  y[i] ~ normal(mu, sigma);
```

and also equivalently,

```
for (i in 1:n) {
  y[i] ~ normal(mu, sigma);
}
```

There's a semicolon after the model statement but not after the brackets indicating the body of the for loop.

In Stan, variables can have names constructed using letters, numbers, and the underscore (`_`) symbol, but nothing else (and a variable name cannot begin with a number). BUGS variables can also include the dot, or period (`.`) symbol.

In Stan, the second argument to the "normal" function is the standard deviation (i.e., the scale), not the variance (as in *Bayesian Data Analysis*) and not the inverse-variance (i.e., precision) (as in BUGS). Thus a normal with mean 1 and standard deviation 2 is `normal(1,2)`, not `normal(1,4)` or `normal(1,0.25)`.

Similarly, the second argument to the "multivariate normal" function is the covariance matrix and not the inverse covariance matrix (i.e., the precision matrix) (as in BUGS). The same is true for the "multivariate student" distribution.

The distributions have slightly different names:

BUGS	Stan
`dnorm`	`normal`
`dbinom`	`binomial`
`dpois`	`poisson`

BUGS	Stan
...	...

Stan, unlike BUGS, allows intermediate quantities, in the form of local variables, to be reassigned. For example, the following is legal and meaningful (if possibly inefficient) Stan code.

```
{
  total = 0;
  for (i in 1:n){
    theta[i] ~ normal(total, sigma);
    total = total + theta[i];
  }
}
```

In BUGS, the above model would not be legal because the variable `total` is defined more than once. But in Stan, the loop is executed in order, so `total` is overwritten in each step.

Stan uses explicit declarations. Variables are declared with base type integer or real, and vectors, matrices, and arrays have specified dimensions. When variables are bounded, we give that information also. For data and transformed parameters, the bounds are used for error checking. For parameters, the constraints are critical to sampling as they determine the geometry over which the Hamiltonian is simulated.

In Stan, variables can be declared as data, transformed data, parameters, transformed parameters, or generated quantities. They can also be declared as local variables within blocks. For more information, see the part of this manual devoted to the Stan programming language and examine at the example models.

Stan allows all sorts of tricks with vector and matrix operations which can make Stan models more compact. For example, arguments to probability functions may be vectorized,[2] allowing

```
for (i in 1:n)
  y[i] ~ normal(mu[i], sigma[i]);
```

to be expressed more compactly as

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

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

The vectorized form is also more efficient because Stan can unfold the computation of the chain rule during algorithmic differentiation.

Stan also allows for arrays of vectors and matrices. For example, in a hierarchical model might have a vector of `K` parameters for each of `J` groups; this can be declared using

```
vector[K] theta[J];
```

Then `theta[j]` is an expression denoting a `K`-vector and may be used in the code just like any other vector variable.

An alternative encoding would be with a two-dimensional array, as in

```
real theta[J,K];
```

The vector version can have some advantages, both in convenience and in computational speed for some operations.

A third encoding would use a matrix:

```
matrix[J,K] theta;
```

but in this case, `theta[j]` is a row vector, not a vector, and accessing it as a vector is less efficient than with an array of vectors. The transposition operator, as in `theta[j]'`, may be used to convert the row vector `theta[j]` to a (column) vector. Column vector and row vector types are not interchangeable everywhere in Stan; see the function signature declarations in the programming language section of this manual.

Stan supports general conditional statements using a standard if-else syntax. For example, a zero-inflated (or -deflated) Poisson mixture model is defined using the if-else syntax as described in the zero inflation section.

Stan supports general while loops using a standard syntax. While loops give Stan full Turing equivalent computational power. They are useful for defining iterative functions with complex termination conditions. As an illustration of their syntax, the for-loop

```
model {
    ....
    for (n in 1:N) {
        ... do something with n ....
    }
}
```

may be recoded using the following while loop.

```
model {
    int n;
    ...
    n = 1;
    while (n <= N) {
        ... do something with n ...
        n = n + 1;
    }
}
```

25.3. Some Differences in the Statistical Models that are Allowed

Stan does not yet support declaration of discrete parameters. Discrete data variables are supported. Inference is supported for discrete parameters as described in the mixture and latent discrete parameters chapters of the manual.

Stan has some distributions on covariance matrices that do not exist in BUGS, including a uniform distribution over correlation matrices which may be rescaled, and the priors based on C-vines defined in Lewandowski, Kurowicka, and Joe (2009). In particular, the Lewandowski et al. prior allows the correlation matrix to be shrunk toward the unit matrix while the scales are given independent priors.

In BUGS you need to define all variables. In Stan, if you declare but don't define a parameter it implicitly has a flat prior (on the scale in which the parameter is defined). For example, if you have a parameter `p` declared as

```
real<lower = 0, upper = 1> p;
```

and then have no sampling statement for `p` in the `model` block, then you are implicitly assigning a uniform $[0, 1]$ prior on `p`.

On the other hand, if you have a parameter `theta` declared with

```
real theta;
```

and have no sampling statement for `theta` in the `model` block, then you are implicitly assigning an improper uniform prior on $(-\infty, \infty)$ to `theta`.

BUGS models are always proper (being constructed as a product of proper marginal and conditional densities). Stan models can be improper. Here is the simplest improper Stan model:

```
parameters {
  real theta;
}
model { }
```

Although parameters in Stan models may have improper priors, we do not want improper *posterior* distributions, as we are trying to use these distributions for Bayesian inference. There is no general way to check if a posterior distribution is improper. But if all the priors are proper, the posterior will be proper also.

Each statement in a Stan model is directly translated into the C++ code for computing the log posterior. Thus, for example, the following pair of statements is legal in a Stan model:

```
y ~ normal(0,1);
y ~ normal(2,3);
```

The second line here does *not* simply overwrite the first; rather, *both* statements contribute to the density function that is evaluated. The above two lines have the effect of including the product, $\mathsf{normal}(y \mid 0, 1) * \mathsf{normal}(y \mid 2, 3)$, into the density function.

For a perhaps more confusing example, consider the following two lines in a Stan model:

```
x ~ normal(0.8 * y, sigma);
y ~ normal(0.8 * x, sigma);
```

At first, this might look like a joint normal distribution with a correlation of 0.8. But it is not. The above are *not* interpreted as conditional entities; rather, they are factors in the joint density. Multiplying them gives, $\mathsf{normal}(x \mid 0.8y, \sigma) \times \mathsf{normal}(y \mid 0.8x, \sigma)$, which is what it is (you can work out the algebra) but it is not the joint distribution where the conditionals have regressions with slope 0.8.

With censoring and truncation, Stan uses the censored-data or truncated-data likelihood—this is not always done in BUGS. All of the approaches to censoring and truncation discussed in Gelman et al. (2013) and Gelman and Hill (2007) may be implemented in Stan directly as written.

Stan, like BUGS, can benefit from human intervention in the form of reparameterization.

25.4. Some Differences when Running from R

Stan can be set up from within R using two lines of code. Follow the instructions for running Stan from R on the Stan web site. You don't need to separately download Stan and RStan. Installing RStan will automatically set up Stan.

In practice we typically run the same Stan model repeatedly. If you pass RStan the result of a previously fitted model the model will not need be recompiled. An example is given on the running Stan from R pages available from the Stan web site.

When you run Stan, it saves various conditions including starting values, some control variables for the tuning and running of the no-U-turn sampler, and the initial random seed. You can specify these values in the Stan call and thus achieve exact replication if desired. (This can be useful for debugging.)

When running BUGS from R, you need to send exactly the data that the model needs. When running RStan, you can include extra data, which can be helpful when playing around with models. For example, if you remove a variable `x` from the model, you can keep it in the data sent from R, thus allowing you to quickly alter the Stan model without having to also change the calling information in your R script.

As in R2WinBUGS and R2jags, after running the Stan model, you can quickly summarize using `plot()` and `print()`. You can access the simulations themselves using various extractor functions, as described in the RStan documentation.

Various information about the sampler, such as number of leapfrog steps, log probability, and step size, is available through extractor functions. These can be useful for understanding what is going wrong when the algorithm is slow to converge.

25.5. The Stan Community

Stan, like WinBUGS, OpenBUGS, and JAGS, has an active community, which you can access via the user's mailing list and the developer's mailing list; see the Stan web site for information on subscribing and posting and to look at archives.

References

Agrawal, Nikhar, Anton Bikineev, Paul Bristow, Marco Guazzone, Christopher Kormanyos, Hubert Holin, Bruno Lalande, et al. 2017. "Double-Exponential Quadrature." https://www.boost.org/doc/libs/1_66_0/libs/math/doc/html/math_toolkit/double_exponen

Aguilar, Omar, and Mike West. 2000. "Bayesian Dynamic Factor Models and Portfolio Allocation." *Journal of Business & Economic Statistics* 18 (3). Taylor & Francis: 338–57.

Ahnert, Karsten, and Mario Mulansky. 2011. "Odeint—Solving Ordinary Differential Equations in C++." *arXiv* 1110.3397.

Albert, J. H., and S. Chib. 1993. "Bayesian Analysis of Binary and Polychotomous Response Data." *Journal of the American Statistical Association* 88: 669–79.

Barnard, John, Robert McCulloch, and Xiao-Li Meng. 2000. "Modeling Covariance Matrices in Terms of Standard Deviations and Correlations, with Application to Shrinkage." *Statistica Sinica*, 1281–1311.

Betancourt, Michael. 2012. "A General Metric for Riemannian Manifold Hamiltonian Monte Carlo." *arXiv* 1212.4693. http://arxiv.org/abs/1212.4693.

Betancourt, Michael, and Mark Girolami. 2013. "Hamiltonian Monte Carlo for Hierarchical Models." *arXiv* 1312.0906. http://arxiv.org/abs/1312.0906.

Blei, David M., and John D. Lafferty. 2007. "A Correlated Topic Model of *Science*." *The Annals of Applied Statistics* 1 (1): 17–37.

Blei, David M., Andrew Y. Ng, and Michael I. Jordan. 2003. "Latent Dirichlet Allocation." *Journal of Machine Learning Research* 3: 993–1022.

Chung, Yeojin, Sophia Rabe-Hesketh, Vincent Dorie, Andrew Gelman, and Jingchen Liu. 2013. "A Nondegenerate Penalized Likelihood Estimator for Variance Parameters in Multilevel Models." *Psychometrika* 78 (4): 685–709.

Clayton, D. G. 1992. "Models for the Analysis of Cohort and Case-Control Studies with Inaccurately Measured Exposures." In *Statistical Models for Longitudinal Studies of Exposure and Health*, edited by James H. Dwyer, Manning Feinleib, Peter Lippert, and Hans Hoffmeister, 301–31. New York: Oxford University Press.

Cohen, Scott D, and Alan C Hindmarsh. 1996. "CVODE, a Stiff/Nonstiff ODE Solver

in C." *Computers in Physics* 10 (2): 138–43.

Cormack, R. M. 1964. "Estimates of Survival from the Sighting of Marked Animals." *Biometrika* 51 (3/4): 429–38.

Curtis, S. McKay. 2010. "BUGS Code for Item Response Theory." *Journal of Statistical Software* 36 (1). American Statistical Association: 1–34.

Dawid, A. P., and A. M. Skene. 1979. "Maximum Likelihood Estimation of Observer Error-Rates Using the EM Algorithm." *Journal of the Royal Statistical Society. Series C (Applied Statistics)* 28 (1): 20–28.

Dempster, A. P., N. M. Laird, and D. B. Rubin. 1977. "Maximum Likelihood from Incomplete Data via the EM Algorithm." *Journal of the Royal Statistical Society. Series B (Methodological)* 39 (1): 1–38.

Dormand, John R, and Peter J Prince. 1980. "A Family of Embedded Runge-Kutta Formulae." *Journal of Computational and Applied Mathematics* 6 (1): 19–26.

Engle, Robert F. 1982. "Autoregressive Conditional Heteroscedasticity with Estimates of Variance of United Kingdom Inflation." *Econometrica* 50: 987–1008.

Fonnesbeck, Chris, Anand Patil, David Huard, and John Salvatier. 2013. *PyMC User's Guide*.

Gelman, Andrew. 2004. "Parameterization and Bayesian Modeling." *Journal of the American Statistical Association* 99: 537–45.

Gelman, Andrew, and Jennifer Hill. 2007. *Data Analysis Using Regression and Multilevel-Hierarchical Models*. Cambridge, United Kingdom: Cambridge University Press.

Gelman, Andrew, J. B. Carlin, Hal S. Stern, David B. Dunson, Aki Vehtari, and Donald B. Rubin. 2013. *Bayesian Data Analysis*. Third. London: Chapman &Hall/CRC Press.

Girolami, Mark, and Ben Calderhead. 2011. "Riemann Manifold Langevin and Hamiltonian Monte Carlo Methods." *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 73 (2): 123–214.

Greene, William H. 2011. *Econometric Analysis*. 7th ed. Prentice-Hall.

Hoeting, Jennifer A., David Madigan, Adrian E Raftery, and Chris T. Volinsky. 1999. "Bayesian Model Averaging: A Tutorial." *Statistical Science* 14 (4): 382–417.

Hoffman, Matthew D., and Andrew Gelman. 2011. "The No-U-Turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo." *arXiv* 1111.4246.

http://arxiv.org/abs/1111.4246.

———. 2014. "The No-U-Turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo." *Journal of Machine Learning Research* 15: 1593–1623. http://jmlr.org/papers/v15/hoffman14a.html.

Jarrett, R. G. 1979. "A Note on the Intervals Between Coal-Mining Disasters." *Biometrika* 66 (1). Biometrika Trust: 191–93.

Kim, Sangjoon, Neil Shephard, and Siddhartha Chib. 1998. "Stochastic Volatility: Likelihood Inference and Comparison with ARCH Models." *Review of Economic Studies* 65: 361–93.

Lambert, Diane. 1992. "Zero-Inflated Poisson Regression, with an Application to Defects in Manufacturing." *Technometrics* 34 (1).

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.

Lincoln, F. C. 1930. "Calculating Waterfowl Abundance on the Basis of Banding Returns." *United States Department of Agriculture Circular* 118: 1–4.

Marsaglia, George. 1972. "Choosing a Point from the Surface of a Sphere." *The Annals of Mathematical Statistics* 43 (2): 645–46.

Neal, Radford M. 1996a. *Bayesian Learning for Neural Networks*. Lecture Notes in Statistics 118. New York: Springer.

———. 1996b. "Sampling from Multimodal Distributions Using Tempered Transitions." *Statistics and Computing* 6 (4): 353–66.

———. 1997. "Monte Carlo Implementation of Gaussian Process Models for Bayesian Regression and Classification." 9702. University of Toronto, Department of Statistics.

———. 2003. "Slice Sampling." *Annals of Statistics* 31 (3): 705–67.

Papaspiliopoulos, Omiros, Gareth O. Roberts, and Martin Sköld. 2007. "A General Framework for the Parametrization of Hierarchical Models." *Statistical Science* 22 (1): 59–73.

Petersen, C. G. J. 1896. "The Yearly Immigration of Young Plaice into the Limfjord from the German Sea." *Report of the Danish Biological Station* 6: 5–84.

Piironen, Juho, and Aki Vehtari. 2016. "Projection Predictive Model Selection for Gaussian Processes." In *Machine Learning for Signal Processing (Mlsp), 2016 Ieee*

26th International Workshop on. IEEE.

Powell, Michael J. D. 1970. "A Hybrid Method for Nonlinear Equations." In *Numerical Methods for Nonlinear Algebraic Equations*, edited by P. Rabinowitz. Gordon; Breach.

Rasmussen, Carl Edward, and Christopher K. I. Williams. 2006. *Gaussian Processes for Machine Learning*. MIT Press.

Richardson, Sylvia, and Walter R. Gilks. 1993. "A Bayesian Approach to Measurement Error Problems in Epidemiology Using Conditional Independence Models." *American Journal of Epidemiology* 138 (6): 430–42.

Rubin, Donald B. 1981. "Estimation in Parallel Randomized Experiments." *Journal of Educational Statistics* 6: 377–401.

Schofield, Matthew R. 2007. "Hierarchical Capture-Recapture Models." PhD thesis, Department of of Statistics, University of Otago, Dunedin.

Serban, Radu, and Alan C Hindmarsh. 2005. "CVODES: The Sensitivity-Enabled ODE Solver in SUNDIALS." In *ASME 2005 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, 257–69. American Society of Mechanical Engineers.

Smith, Teresa C., David J. Spiegelhalter, and Andrew Thomas. 1995. "Bayesian Approaches to Random-Effects Meta-Analysis: A Comparative Study." *Statistics in Medicine* 14 (24): 2685–99.

Swendsen, Robert H., and Jian-Sheng Wang. 1986. "Replica Monte Carlo Simulation of Spin Glasses." *Physical Review Letters* 57: 2607–9.

Warn, David E., S. G. Thompson, and David J. Spiegelhalter. 2002. "Bayesian Random Effects Meta-Analysis of Trials with Binary Outcomes: Methods for the Absolute Risk Difference and Relative Risk Scales." *Statistics in Medicine* 21: 1601–23.

Zellner, Arnold. 1962. "An Efficient Method of Estimating Seemingly Unrelated Regression Equations and Tests for Aggregation Bias." *Journal of the American Statistical Association* 57: 348–68.

Zhang, Hao. 2004. "Inconsistent Estimation and Asymptotically Equal Interpolations in Model-Based Geostatistics." *Journal of the American Statistical Association* 99 (465). Taylor & Francis: 250–61.

Zyczkowski, K., and H.J. Sommers. 2001. "Induced Measures in the Space of Mixed Quantum States." *Journal of Physics A: Mathematical and General* 34 (35). IOP

Publishing: 7111.