Torsten User's Guide

Version 0.91

Torsten Development Team

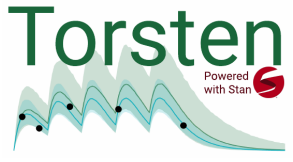


Table of Contents

Overview

This is the official user's guide for Torsten. It provides function references, and model examples for coding statistical models in Torsten.

- Acknowledgements {#acknowledgements.section} This work was funded in part by the following organizations:
- Office of Naval Research (ONR) contract N00014-16-P-2039 provided as part of the Small Business Technology Transfer (STTR) program. The content of the information presented in this document does not necessarily reflect the position or policy of the Government and no official endorsement should be inferred.
- Bill & Melinda Gates Foundation.
- We thank Metrum Research Group, Columbia University, and AstraZeneca. We also thank the Stan Development Team for giving us guidance on how to create new Stan functions and adding features to Stan's core language that facilitate building ODE-based models. We also thank Kyle Baron and Hunter Ford for helpful advice on coding in C++ and using GitHub, Curtis Johnston for reviewing the User Manual, and Yaming Su for using Torsten and giving us feedback.

1. Introduction

Stan is an open source probabilistic programing language designed primarily to do Bayesian data analysis It provides an expressive syntax for statistic modeling and contains an efficient variant of No U-Turn Sampler (NUTS), an adaptative Hamiltonian Monte Carlo algorithm that was proven more efficient than commonly used Monte Carlo Markov Chains (MCMC) samplers for complex high dimensional problems.

Torsten is a collection of Stan functions to facilitate analysis of pharmacometric data. Given an event schedule and an ODE system, it calculates amounts in each compartment. The current version includes [1]:

- Specific linear compartment models:
  - One compartment model with first order absorption.
  - Two compartment model with elimination from and first order absorption into central compartment
- General linear compartment model described by a system of first-order linear Ordinary Differential Equations (ODEs).
  - General compartment model described by a system of first order ODEs.
  - Coupled model with PK forcing function described by a linear one or two compartment model and PD components solved by numerical ODE integration.

The models and data format are based on NONMEM®[2]/NMTRAN/PREDPP conventions including:

- recursive calculation of model predictions, which permits piecewise constant covariate values,
- bolus Or constant rate inputs into any compartment,

[1]The current version of Torsten is a *prototype*. It is being released for review and comment, and to support limited research applications. It has not been rigorously tested and should not be used for critical applications without further testing or cross-checking by comparison with other methods. We encourage interested users to try Torsten out and are happy to assist. Please report issues, bugs, and feature requests on our GitHub page

[2]NONMEM® is licensed and distributed by ICON Development Solutions.

- single dose and multiple dose events,
- steady state dosing events,
- NMTRAN-compartible data items such as TIME, EVID, CMT, AMT, RATE, ADDL, II, and SS.

All real variable arguments in Torsten functions can be passed as Stan `parameters`.

1.1. Implementation summary

- Current Torsten v0.91.0 is based on Stan v2.33.1.
- All functions are programmed in C++ and are compatible with the Stan math automatic differentiation library **?**
- One and two compartment models are based on analytical solutions of governing ODEs.
- General linear compartment models are based on semi-analytical solutions using the built-in matrix exponential function
- General compartment models are solved numerically using built-in ODE integrators in Stan. The tuning parameters of the solver are adjustable. The steady state solution is calculated using a numerical algebraic solver.
- Coupled model that has PK forcing function solved analytically and PD ODE components solved numerically.

2. Installation

Currently Torsten is based on a forked version of Stan and hosted on GitHub. The latest v0.91.0 is compatible with Stan v2.33.1. Torsten can be accessed from command line for `cmdstan` interface and `cmdstanr` for R interface. It requires a modern C++11 compiler as well as a `Make` utility. See `cmdstanr user guide` for details of installation and required toolchain.

2.1. Command line interface

The command line interface `cmdstan` is available along with Torsten and can be found at `Torsten/cmdstan` in the project. Details of installing `cmdstan` can be found in the `cmdstan` user guide.

After installation, one can use the following command to build a Torsten model `model_name` at `model_path`.

```
cd Torsten/cmdstan
make model_path/model_name # linux/macOS
mingw32-make model_path/model_name # windows
```

2.2. R interface

After installing `cmdstanr`, use the following command to set path

```
cmdstanr::set_cmdstan_path("Torsten/cmdstan")
```

Then one can follow `cmdstanr` user's guide to compile and run Torsten models.

2.3. MPI support

Torsten's MPI support is of a different flavour than `reduce_sum` in Stan. MPI parallelisation for Torsten require an MPI library such as

- https://www.mpich.org/downloads/
- https://www.open-mpi.org/software/ompi/

Torsen's implementation is tested on both `MPICH` and `OpenMPI`.

One can enable MPI-supported population/group solvers on both interfaces ^[Since `TORSTEN_MPI` and `STAN_MPI` flags conflict on processes management, they cannot

be used in a same model. ..

MPI support is only available through `cmdstan` interface. To enable it, add/edit `cmdstan/make/local` file with a line `TORSTEN_MPI=1`. One may also need to specify the path to the MPI library, such as

```
CXXFLAGS += -isystem /usr/local/include
CXXFLAGS += -isystem /usr/local/mpich3/include # For Metworx platform
```

Part I

Function references

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 (Andrew 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 observations could be coded as follows.[1]

```
data {
  int<lower=0> N_obs;
  int<lower=0> N_mis;
  array[N_obs] real y_obs;
}
parameters {
  real mu;
  real<lower=0> sigma;
  array[N_mis] real y_mis;
}
model {
```

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

```
  y_obs ~ normal(mu, sigma);
  y_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;
  array[N] vector[2] y;
  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;
  array[N_obs] int<lower=1, upper=N_obs + N_mis> ii_obs;
  array[N_mis] int<lower=1, upper=N_obs + N_mis> ii_mis;
  array[N_obs] real y_obs;
}
transformed data {
  int<lower=0> N = N_obs + N_mis;
}
parameters {
  array[N_mis] real y_mis;
  real<lower=0> sigma;
}
transformed parameters {
```

```
  array[N] real y;
  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 observa-

[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.

tions y and a boolean array y_observed to indicate which values were observed (others can have dummy values in the input).

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

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.

```
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 distribution statements. In transformed data, build up three vectors of indices, for the three cases above:

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

You will need to write functions that pull out the count of observations in each of the three 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;
  array[N] real<upper=U> y;
}
parameters {
  real mu;
  real<lower=0> sigma;
}
model {
  y ~ 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 distribution 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.

```
y ~ normal(mu, sigma) T[L, U];
```

then if any value inside `y` is less than the value of `L` or greater than the value of `U`, the distribution 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 {
  array[N] real<lower=L, upper=U> y;
  // ...
}
```

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;
  array[N] real y;
```

```
}
parameters {
  real<upper=min(y)> L;
  real<lower=max(y)> U;
  real mu;
  real<lower=0> sigma;
}
model {
  L ~ // ...
  U ~ // ...
  y ~ 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;
  array[N_obs] real y_obs;
```

```
  real<lower=max(y_obs)> U;
}
parameters {
  array[N_cens] real<lower=U> y_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 distribution 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

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

where $\Phi()$ is the standard normal cumulative distribution function. This probability is equivalent to the likelihood contribution of knowing that $y_{\text{cens},m} > U$. With M censored observations, the likelihood on the log scale is

$$\begin{aligned}\log \prod_{m=1}^{M} \Pr[y_{\text{cens},m} > U] &= \log\left(1 - \Phi\left(\left(\frac{U-\mu}{\sigma}\right)\right)^M\right) \\ &= M \times \texttt{normal\_lccdf}\left(U \mid \mu, \sigma\right),\end{aligned}$$

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;
  array[N_obs] real y_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 model is used without truncation. The likelihood contribution from the integrated out censored values can not be coded with distribution statement, and the log probability is directly incremented using the calculated log cumulative normal probability of the censored observations.

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;
  array[N_obs] real y_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

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

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(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,

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

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

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

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
  array[N] real y;         // 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 $\theta_k \times \mathsf{normal}\left(y_n \mid \mu_k, \sigma_k\right)$ is calculated and added to the array `lps`. 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

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

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,

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

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]

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.

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

Posterior predictive distribution

Posterior predictive distribution for a new observation $\tilde{y}$ given the complete parameter vector θ will be

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

The normal mixture example from the previous section, with $\theta = (\mu, \sigma, \lambda)$, shows that the model 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

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

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 posterior predictive distribute, 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. Hurdle models can be formulated as combination of point mass at zero and continuous distribution for positive values.

Zero inflation

Consider the following example for zero-inflated Poisson distributions. There is a probability θ of observing a zero, and a probability $1 - \theta$ of observing a count with a $\mathsf{Poisson}(\lambda)$ distribution (now θ is being used for mixing proportions because λ is the traditional notation for a Poisson mean parameter). Given the probability θ and the intensity λ, the distribution for y_n can be written as

$$\begin{array}{ll} y_n = 0 & \text{with probability } \theta, \text{ and} \\ y_n \sim \mathsf{Poisson}(y_n \mid \lambda) & \text{with probability } 1 - \theta. \end{array}$$

Stan does not support conditional distribution statements (with ~) conditional on some parameter, and we need to consider the corresponding likelihood

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

The log likelihood can be coded directly in Stan (with `target +=`) as follows.

```
data {
  int<lower=0> N;
  array[N] int<lower=0> y;
}
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(log(theta),
                            log1m(theta)
                              + poisson_lpmf(y[n] | lambda));
    } else {
      target += log1m(theta)
                  + poisson_lpmf(y[n] | lambda);
    }
  }
}
```

The `log1m(theta)` computes `log(1-theta)`, but is more computationally stable. 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 computationally 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 | 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(array[] 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);
  array[N - N_zero] int<lower=1> y_nonzero;
  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(log(theta),
                        log1m(theta)
                          + poisson_lpmf(0 | lambda));
   target += N_nonzero * log1m(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. Given the probability θ and the intensity λ, the distribution for y_n can be written as [

$$\begin{aligned} y_n &= 0 \quad \text{with probability } \theta, \text{ and} \\ y_n &\sim \text{Poisson}_{x \neq 0}(y_n \mid \lambda) \quad \text{with probability } 1 - \theta, \end{aligned}$$

] Where $\text{Poisson}_{x \neq 0}$ is a truncated Poisson distribution, truncated at 0.

The corresponding likelihood function for the hurdle model is defined by

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

where PoissonCDF is the cumulative distribution function for the Poisson distribution and and $1 - \text{PoissonCDF}(0 \mid \lambda)$ is the relative normalization term for the truncated Poisson (truncated at 0).

The hurdle model is even more straightforward to program in Stan, as it does not require an explicit mixture.

```
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

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

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

The model block is then 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 Andrew 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 {
  // ...
  array[N] real x_meas;   // measurement of x
  real<lower=0> tau;      // measurement noise
}
parameters {
  array[N] real x;    // 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) by Andrew 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

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

The rounding process entails that $z_n \in (y_n - 0.5, y_n + 0.5)$[1]. The probability mass function for the discrete observation y is then given by marginalizing out the unrounded measurement, producing the likelihood

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

[1] There are several different rounding rules (see, e.g., Wikipedia: Rounding), which affect which interval ends are open and which are closed, but these do not matter here as for continuous z_n $p(z_n = y_n - 0.5) = p(z_n = y_n + 0.5) = 0$.

Gelman's answer for this problem took the noninformative prior to be uniform in the variance σ^2 on the log scale, but we replace it with more recently recommended half-normal prior on σ

$$\sigma \sim \text{normal}^+(0, 1).$$

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

$$\begin{aligned} p(\mu, \sigma \mid y) &\propto p(\mu, \sigma)\, p(y \mid \mu, \sigma) \\ &\propto \text{normal}^+(\sigma \mid 0, 1) \prod_{n=1}^{5} \left(\Phi\left(\frac{y_n + 0.5 - \mu}{\sigma} \right) - \Phi\left(\frac{y_n - 0.5 - \mu}{\sigma} \right) \right). \end{aligned}$$

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;
}
model {
  sigma ~ normal(0, 1);
  for (n in 1:N) {
    target += log_diff_exp(normal_lcdf(y[n] + 0.5 | mu, sigma),
                           normal_lcdf(y[n] - 0.5 | mu, sigma));
  }
}
```

where `normal_lcdf(y[n]+0.5 | mu, sigma)` is equal to `log(Phi((y[n] + 0.5 - mu) / sigma))`, and `log_diff_exp(a, b)` computes `log(exp(a) - exp(b))` in numerically more stable way.

Alternatively, the model may be defined with latent parameters for the unrounded measurements z_n. The Stan code in this case uses a distribution statement for z_n directly while respecting the constraint $z_n \in (y_n - 0.5, y_n + 0.5)$.

```
data {
  int<lower=0> N;
```

```
  vector[N] y;
}
parameters {
  real mu;
  real<lower=0> sigma;
  vector<lower=y-0.5, upper=y+0.5>[N] z;
}
model {
  sigma ~ normal(0, 1);
  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 (Andrew Gelman et al. 2013, sec. 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.[2]

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

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

The log odds and standard errors can be defined in a transformed data block, though care must be taken not to use integer division.[3]

```
transformed data {
  array[J] real y;
  array[J] real<lower=0> sigma;
  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])
```

[2] 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.

[3] 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.

```
                    + 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 distribution 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 $\text{normal}(y \mid \mu, \sigma) = \text{normal}(\mu \mid y, \sigma)$.

Hierarchical model

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

```
parameters {
  array[J] real theta;  // 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 distribution statement for `y` appears unchanged, the parameter `theta` is now a vector. The distribution statement for `theta` is also vectorized, with the hyperparameters `mu` and `tau` themselves being given wide priors compared to the scale of the data.

Donald B. 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 Andrew Gelman et al. (2013, sec. 19.4) provide 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, sec. 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

[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.

point in time s. The full model expressed using a latent discrete parameter s is

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

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,

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

[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.

where the likelihood is defined by marginalizing s as

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

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;
  array[T] int<lower=0> D;
}
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 `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.

Figure 7.1: Analytical change-point posterior

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

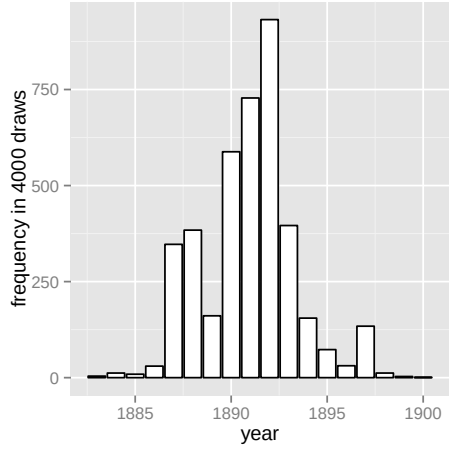


Figure 7.2: Sampled change-point posterior

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 second 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 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`, 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,

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

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).

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 {
  array[7] int<lower=0> history;
}
parameters {
  array[2] real<lower=0, upper=1> phi;
  array[3] real<lower=0, upper=1> p;
}
transformed parameters {
  array[2] real<lower=0, upper=1> chi;
  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;
  array[I, T] int<lower=0, upper=1> y;
}
```

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(array[] int y_i) {
    for (k in 1:size(y_i)) {
      if (y_i[k]) {
        return k;
      }
    }
    return 0;
  }
  int last_capture(array[] 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;
      }
    }
    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 {
  array[I] int<lower=0, upper=T> first;
  array[I] int<lower=0, upper=T> last;
  vector<lower=0, upper=I>[T] n_captured;
  for (i in 1:I) {
    first[i] = first_capture(y[i]);
  }
```

[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.

[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.

```
  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 {
  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 be 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.

A. P. 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:

- 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

[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.

[7] In the subscript, z_i is written as $z[i]$ to improve legibility.

to Dirichlet priors,

$$\pi \sim \text{Dirichlet}(\alpha)$$

and

$$\theta_{j,k} \sim \text{Dirichlet}(\beta_k)$$

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.

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

$$p(z, y \mid \theta, \pi) = \prod_{i=1}^{I} \left(\text{categorical}(z_i \mid \pi) \prod_{j=1}^{J} \text{categorical}(y_{i,j} \mid \theta_{j,z[i]}) \right).$$

This can be done item by item, with

$$p(y \mid \theta, \pi) = \prod_{i=1}^{I} \sum_{k=1}^{K} \left(\text{categorical}(k \mid \pi) \prod_{j=1}^{J} \text{categorical}(y_{i,j} \mid \theta_{j,k}) \right).$$

In the missing data model, only the observed labels would be used in the inner product.

A. P. 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,

$$\log p(y \mid \theta, \pi) = \sum_{i=1}^{I} \log \left(\sum_{k=1}^{K} \exp \left(\log \text{categorical}(k \mid \pi) \; + \; \sum_{j=1}^{J} \log \text{categorical}(y_{i,j} \mid \theta_{j,k}) \right) \right)$$

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 (A. P. Dawid and Skene 1979).

```
data {
  int<lower=2> K;
  int<lower=1> I;
  int<lower=1> J;

  array[I, J] int<lower=1, upper=K> y;

  vector<lower=0>[K] alpha;
  vector<lower=0>[K] beta[K];
}
parameters {
  simplex[K] pi;
  array[J, K] simplex[K] theta;
}
transformed parameters {
  array[I] vector[K] log_q_z;
  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 A. P. 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.

7.5. The mathematics of recovering marginalized parameters

Introduction

This section describes in more detail the mathematics of statistical inference using the output of marginalized Stan models, such as those presented in the last three sections. It provides a mathematical explanation of why and how certain manipulations of Stan's output produce valid summaries of the posterior distribution when discrete parameters have been marginalized out of a statistical model. Ultimately, however, fully understanding the mathematics in this section is *not* necessary to fit models with discrete parameters using Stan.

Throughout, the model under consideration consists of both continuous parameters, Θ, and discrete parameters, Z. It is also assumed that Z can only take finitely many values, as is the case for all the models described in this chapter of the User's Guide. To simplify notation, any conditioning on data is suppressed in this section, except where specified. As with all Bayesian analyses, however, all inferences using models with marginalized parameters are made conditional on the observed data.

Estimating expectations

When performing Bayesian inference, interest often centers on estimating some (constant) low-dimensional summary statistics of the posterior distribution. Mathematically, we are interested in estimating μ, say, where $\mu = \mathbb{E}[g(\Theta, Z)]$ and $g(\cdot)$ is an arbitrary function. An example of such a quantity is $\mathbb{E}[\Theta]$, the posterior mean of the continuous parameters, where we would take $g(\theta, z) = \theta$. To estimate μ the most common approach is to sample a series of values, at least approximately, from the posterior distribution of the parameters of interest. The numerical values of these draws can then be used to calculate the quantities of interest. Often, this process of calculation is trivial, but more care is required when working with marginalized posteriors as we describe in this section.

If both Θ and Z were continuous, Stan could be used to sample M draws from the joint posterior $p_{\Theta,Z}(\theta, z)$ and then estimate μ with

$$\hat{\mu} = \frac{1}{M} \sum_{i=1}^{M} g(\theta^{(i)}, z^{(i)}).$$

Given Z is discrete, however, Stan cannot be used to sample from the joint posterior (or even to do optimization). Instead, as outlined in the previous sections describing specific models, the user can first marginalize out Z from the joint posterior to give the marginalized posterior $p_{\Theta}(\theta)$. This marginalized posterior can then be implemented in Stan as usual, and Stan will give draws $\{\theta^{(i)}\}_{i=1}^{M}$ from the marginalized posterior.

Using only these draws, how can we estimate $\mathbb{E}[g(\Theta, Z)]$? We can use a conditional estimator. We explain in more detail below, but at a high level the idea is that, for each function g of interest, we compute

$$h(\Theta) = \mathbb{E}[g(\Theta, Z) \mid \Theta]$$

and then estimate $\mathbb{E}[g(\Theta, Z)]$ with

$$\hat{\mu} = \frac{1}{M} \sum_{i=1}^{M} h(\theta^{(i)}).$$

This estimator is justified by the law of iterated expectation, the fact that

$$\mathbb{E}[h(\Theta)] = \mathbb{E}[\mathbb{E}[g(\Theta, Z)] \mid \Theta] = \mathbb{E}[g(\Theta, Z)] = \mu.$$

Using this marginalized estimator provides a way to estimate the expectation of any function $g(\cdot)$ for all combinations of discrete or continuous parameters in the model. However, it presents a possible new challenge: evaluating the conditional expectation $\mathbb{E}[g(\Theta, Z) \mid \Theta]$.

Evaluating the conditional expectation

Fortunately, the discrete nature of Z makes evaluating $\mathbb{E}[g(\Theta, Z) \mid \Theta]$ easy. The function $h(\Theta)$ can be written as:

$$h(\Theta) = \mathbb{E}[g(\Theta, Z) \mid \Theta] = \sum_k g(\Theta, k) \Pr[Z = k \mid \Theta],$$

where we sum over the possible values of the latent discrete parameters. An essential part of this formula is the probability of the discrete parameters conditional on the continuous parameters, $\Pr[Z = k \mid \Theta]$. More detail on how this quantity can be calculated is included below. Note that if Z takes infinitely many values then computing the infinite sums will involve, potentially computationally expensive, approximation.

When $g(\theta, z)$ is a function of either θ or z only, the above formula simplifies further.

In the first case, where $g(\theta, z) = g(\theta)$, we have:

$$\begin{aligned} h(\Theta) &= \sum_k g(\Theta) \Pr[Z = k \mid \Theta] \\ &= g(\Theta) \sum_k \Pr[Z = k \mid \Theta] \\ &= g(\Theta). \end{aligned}$$

This means that we can estimate $\mathbb{E}[g(\Theta)]$ with the standard, seemingly unconditional, estimator:

$$\frac{1}{M} \sum_{i=1}^{M} g(\theta^{(i)}).$$

Even after marginalization, computing expectations of functions of the continuous parameters can be performed as if no marginalization had taken place.

In the second case, where $g(\theta, z) = g(z)$, the conditional expectation instead simplifies as follows:

$$h(\Theta) = \sum_k g(k) \Pr[Z = k \mid \Theta].$$

An important special case of this result is when $g(\theta, z) = \mathrm{I}(z = k)$, where I is the indicator function. This choice allows us to recover the probability mass function of the discrete random variable Z, since $\mathbb{E}[\mathrm{I}(Z = k)] = \Pr[Z = k]$. In this case,

$$h(\Theta) = \sum_k \mathrm{I}(z = k) \Pr[Z = k \mid \Theta] = \Pr[Z = k \mid \Theta].$$

The quantity $\Pr[Z = k]$ can therefore be estimated with:

$$\frac{1}{M} \sum_{i=1}^{M} \Pr[Z = k \mid \Theta = \theta^{(i)}].$$

When calculating this conditional probability it is important to remember that we are also conditioning on the observed data, Y. That is, we are really estimating $\Pr[Z = k \mid Y]$ with

$$\frac{1}{M} \sum_{i=1}^{M} \Pr[Z = k \mid \Theta = \theta^{(i)}, Y].$$

This point is important as it suggests one of the main ways of calculating the required conditional probability. Using Bayes's theorem gives us

$$\Pr[Z = k \mid \Theta = \theta^{(i)}, Y] = \frac{\Pr[Y \mid Z = k, \Theta = \theta^{(i)}] \Pr[Z = k \mid \Theta = \theta^{(i)}]}{\sum_{k=1}^{K} \Pr[Y \mid Z = k, \Theta = \theta^{(i)}] \Pr[Z = k \mid \Theta = \theta^{(i)}]}.$$

Here, $\Pr[Y \mid \Theta = \theta^{(i)}, Z = k]$ is the likelihood conditional on a particular value of the latent variables. Crucially, all elements of the expression can be calculated using the draws from the posterior of the continuous parameters and knowledge of the model structure.

Other than the use of Bayes's theorem, $\Pr[Z = k \mid \theta = \theta^{(i)}, Y]$ can also be extracted by coding the Stan model to include the conditional probability explicitly (as is done for the Dawid–Skene model).

For a longer introduction to the mathematics of marginalization in Stan, which also covers the connections between Rao–Blackwellization and marginalization, see Pullin, Gurrin, and Vukcevic (2021).

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;
  array[J, K] int<lower=0, upper=1> y;
  // ...
model {
  for (j in 1:J) {
    for (k in 1:K) {
      y[j, k] ~ bernoulli_logit(delta[k] * (alpha[j] - beta[k]));
    }
  }
  // ...
}
```

When not every student is given every question, the dense array coding will no longer work, because Stan does not support undefined values.

The following missing data example shows an example with $J = 3$ and $K = 4$, with missing responses shown as NA, as in R.

$$y = \begin{bmatrix} 0 & 1 & \text{NA} & 1 \\ 0 & \text{NA} & \text{NA} & 1 \\ \text{NA} & 0 & \text{NA} & \text{NA} \end{bmatrix}$$

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 indices along with the value. With columns *jj* and *kk* used for the indexes (following Andrew Gelman and Hill (2007)), the 2-D array y is recoded as a table. The number of rows in the table equals the number of defined array elements, here $y_{1,1} = 0$, $y_{1,2} = 1$, up to $y_{3,2} = 1$. As the array becomes larger and sparser, the long form becomes the more economical encoding.

jj	*kk*	y
1	1	0
1	2	1
1	4	1
2	1	0
2	4	1
3	2	0

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;
  array[N] int<lower=1, upper=J> jj;
  array[N] int<lower=1, upper=K> kk;
  array[N] int<lower=0, upper=1> y;
  // ...
}
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 z 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

$$\sum_{n=1}^{3} \log \text{normal}(y_n \mid \mu_n, \sigma).$$

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
```

```
  array[K] int s;   // 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,

$$z_n \sim \text{categorical}(1/K),$$

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,

$$y_n \sim \text{normal}(\mu_{z[n]}, \Sigma_{z[n]})$$

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

$$\text{normal}\left(y_n \mid \mu_k, \text{diag\_matrix}(\mathbf{1})\right) \propto \exp\left(-\frac{1}{2}\sum_{d=1}^{D}(\mu_{k,d} - y_{n,d})^2\right).$$

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
  array[N] vector[D] y;   // observations
}
```

```
transformed data {
  real<upper=0> neg_log_K;
  neg_log_K = -log(K);
}
parameters {
  array[K] vector[D] mu; // cluster means
}
transformed parameters {
  array[N, K] real<upper=0> soft_z; // 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
  array[M] int<lower=1, upper=K> z;    // topic for doc m
```

```
  array[N] int<lower=1, upper=V> w;    // word n
  array[N] int<lower=1, upper=M> doc;  // 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
  array[K] simplex[V] phi;      // 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

$$
\begin{aligned}
&\log p(w_{m,1},\ldots,w_{m,N_m} \mid \theta,\phi) \\
&\quad = \log \sum_{k=1}^{K} \left(\text{categorical}(k \mid \theta) \times \prod_{n=1}^{N_m} \text{categorical}(w_{m,n} \mid \phi_k) \right) \\
&\quad = \log \sum_{k=1}^{K} \exp \left(\log \text{categorical}(k \mid \theta) + \sum_{n=1}^{N_m} \log \text{categorical}(w_{m,n} \mid \phi_k) \right).
\end{aligned}
$$

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 {
  array[M, K] real gamma;
  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 | theta);
    }
  }
  for (n in 1:N) {
    for (k in 1:K) {
      gamma[doc[n], k] = gamma[doc[n], k]
                         + categorical_lpmf(w[n] | phi[k]);
    }
  }
  for (m in 1:M) {
    target += log_sum_exp(gamma[m]);
  }
}
```

The local variable `gamma[m, k]` represents the value

$$
\gamma_{m,k} = \log \text{categorical}(k \mid \theta) + \sum_{n=1}^{N_m} \log \text{categorical}(w_{m,n} \mid \phi_k).
$$

Given γ, the posterior probability that document m is assigned category k is

$$\Pr[z_m = k \mid w, \alpha, \beta] = \exp\left(\gamma_{m,k} - \log \sum_{k=1}^{K} \exp\left(\gamma_{m,k}\right)\right).$$

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 Andrew Gelman et al. (2013) and Andrew 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 generalizes 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,

$$\begin{aligned}
&\log p(\theta, \phi \mid w, \alpha, \beta) \\
&\quad = \sum_{m=1}^{M} \log \text{Dirichlet}(\theta_m \mid \alpha) + \sum_{k=1}^{K} \log \text{Dirichlet}(\phi_k \mid \beta) \\
&\qquad + \sum_{m=1}^{M} \sum_{n=1}^{N[m]} \log \left(\sum_{z=1}^{K} \text{categorical}(z \mid \theta_m) \times \text{categorical}(w_{m,n} \mid \phi_z) \right)
\end{aligned}$$

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
  array[N] int<lower=1, upper=V> w;    // word n
  array[N] int<lower=1, upper=M> doc;  // doc ID for word n
  vector<lower=0>[K] alpha;     // topic prior
  vector<lower=0>[V] beta;      // word prior
}
parameters {
  array[M] simplex[K] theta;    // topic dist for doc m
  array[K] simplex[V] phi;      // 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) {
    array[K] real gamma;
    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 {
  array[K] simplex[V] phi;     // word dist for topic k
  array[M] vector[K] eta;      // topic dist for doc m
}
transformed parameters {
  array[M] simplex[K] theta;
  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
  array[M] vector[K] eta;    // logit topic dist for doc m
  array[K] simplex[V] phi;   // 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

$$\text{LkjCorr}(\Omega \mid \alpha) \propto \det(\Omega)^{\alpha - 1}$$

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 or by using the specialized covariance functions outlined below, and plugging the result into the Gaussian model.
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).

Note that fitting Gaussian processes as described below using exact inference by computing Cholesky of the covariance matrix scales cubicly with the size of data. Due to how Stan autodiff is implemented, Stan is also slower than Gaussian process

specialized software. It is likely that Gaussian processes using exact inference by computing Cholesky of the covariance matrix with $N > 1000$ are too slow for practical purposes in Stan. There are many approximations to speed-up Gaussian process computation, from which the basis function approaches for 1-3 dimensional x are easiest to implement in Stan (see, e.g., Riutort-Mayol et al. (2023)).

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.

[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.

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 *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;
  array[N] real x;
}
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;
  array[N] real x;
}
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;
  array[N] vector[D] x;
}
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

$$\eta \sim \text{normal}(\mathbf{0}, \mathbf{1}),$$

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,

$$\mu + L\eta \sim \text{multivariate normal}(\mu(x), K(x \mid \theta)).$$

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;
  array[N] real x;
  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 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 distribution, 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 in-

ference 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;
  array[N] real x;
  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 $\text{normal}(0, 1)$ prior on `eta` like we did in the Cholesky-parameterized GP in the simulation section. The second difference is that although we could code the distribution statement for y with one N-dimensional multivariate normal with an identity covariance matrix multiplied by σ^2, we instead use vectorized univariate normal distribution, which is equivalent but more efficient to use.

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 Poisson distribution with log link function, 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 {
  // ...
  array[N] int<lower=0> y;
  // ...
}
// ...
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 {
  // ...
  array[N] int<lower=0, upper=1> z;
  // ...
}
// ...
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" by Neal (1996a), but this is misleading, because the magnitude of 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(array[] 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;
```

```
  array[N] vector[D] x;
  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 begin 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.[2].

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

[2]A boundary-avoiding prior is just one where the limit of the density is zero at the boundary, the result of which is estimates that are pushed away from the boundary.

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:

$$f(x \mid a, b, p) = \frac{(a/b)^{p/2}}{2K_p\left(\sqrt{ab}\right)} x^{p-1} \exp\left(-(ax + b/x)/2\right)$$
$$x, a, b \in \mathbb{R}^+, \quad p \in \mathbb{Z}$$

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;
  array[N1] real x1;
  vector[N1] y1;
  int<lower=1> N2;
  array[N2] real x2;
}
transformed data {
  real delta = 1e-9;
  int<lower=1> N = N1 + N2;
  array[N] real x;
  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 distribution statement for y.

The generated quantities block defines the quantity y2. We generate y2 by randomly generating N2 values from 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;
  array[N1] real x1;
  array[N1] int<lower=0, upper=1> z1;
  int<lower=1> N2;
  array[N2] real x2;
}
transformed data {
  real delta = 1e-9;
  int<lower=1> N = N1 + N2;
  array[N] real x;
  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 {
  array[N2] int z2;
  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,

$$p\left(\tilde{y} \mid \tilde{x}, y, x\right) = \text{normal}\left(K^{\top}\Sigma^{-1}y,\ \Omega - K^{\top}\Sigma^{-1}K\right),$$

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 number of matrix-matrix multiplications when computing the conditional mean and the conditional covariance of $p(\tilde{y})$.

```
functions {
  vector gp_pred_rng(array[] real x2,
                     vector y1,
                     array[] 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;
  array[N1] real x1;
  vector[N1] y1;
  int<lower=1> N2;
  array[N2] real x2;
}
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:

$$
\begin{aligned}
y_i &\sim \text{multivariate normal}\left(f(x_i), \mathbf{I}_M \sigma^2\right) \\
f(x) &\sim \text{GP}\left(m(x), K(x \mid \theta, \phi)\right) \\
K(x \mid \theta) &\in \mathbb{R}^{M \times M}, \quad f(x), m(x) \in \mathbb{R}^M
\end{aligned}
$$

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;
  array[N] real x;
  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 it 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 Muller (1959). 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. Muller (1959) 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.

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 systems of algebraic equations. These systems can be solved either with the Newton method, as implemented in the Kinsol package (Hindmarsh et al. 2005), or with the Powell hybrid method (Powell 1970). The function signatures for Stan's algebraic solvers 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
              data array[] real x_r, // data (real)
              array[] 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]';
  array[0] real x_r;
  array[0] int x_i;
```

```
}

transformed parameters {
  vector[2] theta = [3, 6]';
  vector[2] y;

  y = solve_newton(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 previously uses the default control settings. The `_tol` variant of the solver function allows three additional parameters, all of

which must be supplied.

```
y = solve_newton_tol(system, y_guess, theta, x_r, x_i,
                     scaling_step, f_tol, max_steps);
```

For the Newton solver the three control arguments are scaling step, function tolerance, and maximum number of steps. For the Powell's hybrid method the three control arguments are relative tolerance, function tolerance, and maximum number of steps. If a Newton step is smaller than the scaling step tolerance, the code breaks, assuming the solver is no longer making significant progress. If set to 0, this constraint is ignored. For Powell's hybrid method the relative tolerance is the estimated relative error of the solver and serves to test if a satisfactory solution has been found. After convergence of the either solver, the proposed solution is plugged into the algebraic system and its norm is compared to the function tolerance. If the norm is below the function tolerance, the solution is deemed acceptable. If the solver solver reaches the maximum number of steps, it stops and returns an error message. If one of the criteria 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 `scaling_step = 1e-3` (10^{-3}), `rel_tol = 1e-10` (10^{-10}), `f_tol = 1e-6` (10^{-6}), and `max_steps = 200` (200).

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. The sensitivity can be analysed using importance sampling without need to re-run MCMC with different tolerances as shown by Timonen et al. (2023).

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 number of different methods for solving systems of ordinary differential equations (ODEs). All of these methods adaptively refine their solutions in order to satisfy given tolerances, but internally they handle calculations quite a bit differently.

Because Stan's algorithms requires gradients of the log density, the ODE solvers must not only provide the solution to the ODE itself, but also the gradient of the ODE solution with respect to parameters (the sensitivities). Two fundamentally different approaches are available in Stan to solve this problem, each having very different computational cost depending on the number of ODE states N and the number of parameters M being used:

- A *forward sensitivity* solver expands the base ODE system with additional ODE equations for the gradients of the solution. For each parameter, an additional full set of N sensitivity states are added meaning that the full ODE solved has $N + N \cdot M$ states.
- An *adjoint sensitivity* solver starts by solving the base ODE system forward in time to get the ODE solution and then solves another ODE system (the adjoint) backward in time to get the gradients. The forward and reverse solves both have N states each. There is additionally one quadrature problem solved for every parameter.

The adjoint sensitivity approach scales much better than the forward sensitivity approach. Whereas the computational cost of the forward approach scales multiplicatively in the number of ODE states N and parameters M, the adjoint sensitivity approach scales linear in states N and parameters M. However, the adjoint problem is harder to configure and the overhead for small problems actually makes it slower than solving the full forward sensitivity system. With that in mind, the rest of this introduction focuses on the forward sensitivity interfaces. For information on the adjoint sensitivity interface see the Adjoint ODE solver

Two interfaces are provided for each forward sensitivity solver: one with default tolerances and default max number of steps, and one that allows these controls to be modified. Choosing tolerances is important for making any of the solvers work well – the defaults will not work everywhere. The tolerances should be chosen primarily with consideration to the scales of the solutions, the accuracy needed

for the solutions, and how the solutions are used in the model. For instance, if a solution component slowly varies between 3.0 and 5.0 and measurements of the ODE state are noisy, then perhaps the tolerances do not need to be as tight as for a situation where the solutions vary between 3.0 and 3.1 and very high precision measurements of the ODE state are available. It is also often useful to reduce the absolute tolerance when a component of the solution is expected to approach zero. For information on choosing tolerances, see the control parameters section.

The advantage of adaptive solvers is that as long as reasonable tolerances are provided and an ODE solver well-suited to the problem is chosen the technical details of solving the ODE can be abstracted away. The catch is that it is not always clear from the outset what reasonable tolerances are or which ODE solver is best suited to a problem. In addition, as changes are made to an ODE model, the optimal solver and tolerances may change.

With this in mind, the four forward solvers are `rk45`, `bdf`, `adams`, and `ckrk`. If no other information about the ODE is available, start with the `rk45` solver. The list below has information on when each solver is useful.

If there is any uncertainty about which solver is the best, it can be useful to measure the performance of all the interesting solvers using `profile` statements. It is difficult to always know exactly what solver is the best in all situations, but a `profile` can provide a quick check.

- `rk45`: a fourth and fifth order Runge-Kutta method for non-stiff systems (Dormand and Prince 1980; Ahnert and Mulansky 2011). `rk45` is the most generic solver and should be tried first.
- `bdf`: a variable-step, variable-order, backward-differentiation formula implementation for stiff systems (Cohen and Hindmarsh 1996; Serban and Hindmarsh 2005). `bdf` is often useful for ODEs modeling chemical reactions.
- `adams`: a variable-step, variable-order, Adams-Moulton formula implementation for non-stiff systems (Cohen and Hindmarsh 1996; Serban and Hindmarsh 2005). The method has order up to 12, hence is commonly used when high-accuracy is desired for a very smooth solution, such as in modeling celestial mechanics and orbital dynamics (Montenbruck and Gill 2000).
- `ckrk`: a fourth and fifth order explicit Runge-Kutta method for non-stiff and semi-stiff systems (Cash and Karp 1990; Mazzia, Cash, and Soetaert 2012). The difference between `ckrk` and `rk45` is that `ckrk` should perform better for systems that exhibit rapidly varying solutions. Often in those situations the

derivatives become large or even nearly discontinuous, and `ckrk` is designed to address such problems.

For a discussion of stiff ODE systems, see the stiff ODE section. For information on the adjoint sensitivity interface see the Adjoint ODE solver section. The function signatures for Stan's ODE solvers can be found in the function reference manual section on ODE solvers.

13.1. Notation

An ODE is defined by a set of differential equations, $y(t,\theta)' = f(t,y,\theta)$, and initial conditions, $y(t_0,\theta) = y_0$. The function $f(t,y,\theta)$ is called the system function. The θ dependence is included in the notation for $y(t,\theta)$ and $f(t,y,\theta)$ as a reminder that the solution is a function of any parameters used in the computation.

13.2. Example: simple harmonic oscillator

As an example of a system of ODEs, consider a harmonic oscillator. In a harmonic oscillator a particle disturbed from equilibrium is pulled back towards its equilibrium position by a force proportional to its displacement from equilibrium. The system here additionally has a friction force proportional to particle speed which points in the opposite direction of the particle velocity. The system state will be a pair $y = (y_1, y_2)$ representing position and speed. The change in the system with respect to time is given by the following differential equations.[1]

$$\frac{d}{dt} y_1 = y_2$$
$$\frac{d}{dt} y_2 = -y_1 - \theta y_2$$

The state equations implicitly defines the state at future times as a function of an initial state and the system parameters.

13.3. Coding the ODE system function

The first step in coding an ODE system in Stan is defining the ODE system function. The system functions require a specific signature so that the solvers know how to use them properly.

[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` and `ckrk` solver.

The first argument to the system function is time, passed as a `real`; the second argument to the system function is the system state, passed as a `vector`, and the return value from the system function are the current time derivatives of the state defined as a `vector`. Additional arguments can be included in the system function to pass other information into the solve (these will be passed through the function that starts the ODE integration). These argument can be parameters (in this case, the friction coefficient), data, or any quantities that are needed to define the differential equation.

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).

```
vector sho(real t,        // time
           vector y,      // state
           real theta) {  // friction parameter
  vector[2] dydt;
  dydt[1] = y[2];
  dydt[2] = -y[1] - theta * y[2];
  return dydt;
}
```

The function takes in a time `t` (a `real`), the system state `y` (a `vector`), and the parameter `theta` (a `real`). The function returns a `vector` of time derivatives of the system state at time `t`, state `y`, and parameter `theta`. The simple harmonic oscillator coded here does not have time-sensitive equations; that is, `t` does not show up in the definition of `dydt`, however it is still required.

Strict signature

The types in the ODE system function are strict. The first argument is the time passed as a `real`, the second argument is the state passed as a `vector`, and the return type is a `vector`. A model that does not have this signature will fail to compile. The third argument onwards can be any type, granted all the argument types match the types of the respective arguments in the solver call.

All of these are possible ODE signatures:

```
vector myode1(real t, vector y, real a0);
vector myode2(real t, vector y, array[] int a0, vector a1);
vector myode3(real t, vector y, matrix a0, array[] real a1, row_vector a2)
```

but these are not allowed:

```
vector myode1(real t, array[] real y, real a0);
// Second argument is not a vector
array[] real myode2(real t, vector y, real a0);
// Return type is not a vector
vector myode3(vector y, real a0);
// First argument is not a real and second is not a vector
```

13.4. Measurement error models

Noisy observations of the ODE state can be used to estimate the parameters and/or the initial state of the system.

Simulating noisy measurements

As an example, suppose the simple harmonic oscillator has a parameter value of $\theta = 0.15$ and an initial state $y(t = 0, \theta = 0.15) = (1, 0)$. Assume the system is measured at 10 time points, $t = 1, 2, \cdots, 10$, where each measurement of $y(t, \theta)$ has independent $\text{normal}(0, 0.1)$ error in both dimensions ($y_1(t, \theta)$ and $y_2(t, \theta)$).

The following model can be used to generate data like this:

```
functions {
  vector sho(real t,
             vector y,
             real theta) {
    vector[2] dydt;
    dydt[1] = y[2];
    dydt[2] = -y[1] - theta * y[2];
    return dydt;
  }
}
data {
  int<lower=1> T;
  vector[2] y0;
  real t0;
  array[T] real ts;
  real theta;
}
model {
}
generated quantities {
  array[T] vector[2] y_sim = ode_rk45(sho, y0, t0, ts, theta);
```

```
  // add measurement error
  for (t in 1:T) {
    y_sim[t, 1] += normal_rng(0, 0.1);
    y_sim[t, 2] += normal_rng(0, 0.1);
  }
}
```

The system parameters `theta` and initial state `y0` are read in as data along with the initial time `t0` and observation times `ts`. The ODE is solved for the specified times, and then random measurement errors are added to produce simulated observations `y_sim`. Because the system is not stiff, the `ode_rk45` solver is used.

This program illustrates the way in which the ODE solver is called in a Stan program,

```
array[T] vector[2] y_sim = ode_rk45(sho, y0, t0, ts, theta);
```

this returns the solution of the ODE initial value problem defined by system function `sho`, initial state `y0`, initial time `t0`, and parameter `theta` at the times `ts`. The call explicitly specifies the non-stiff RK45 solver.

The parameter `theta` is passed unmodified to the ODE system function. If there were additional arguments that must be passed, they could be appended to the end of the ode call here. For instance, if the system function took two parameters, θ and β, the system function definition would look like:

```
vector sho(real t, vector y, real theta, real beta) { ... }
```

and the appropriate ODE solver call would be:

```
ode_rk45(sho, y0, t0, ts, theta, beta);
```

Any number of additional arguments can be added. They can be any Stan type (as long as the types match between the ODE system function and the solver call).

Because all none of the input arguments are a function of parameters, the ODE solver is called in the generated quantities block. The random measurement noise is added to each of the `T` outputs with `normal_rng`.

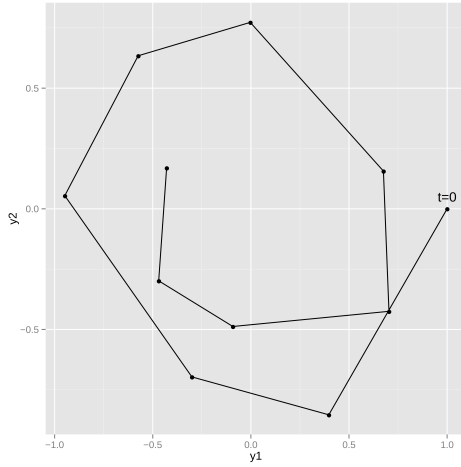


Figure 13.1: Typical realization of harmonic oscillator trajectory.

Estimating system parameters and initial state

These ten noisy observations of the state can be used to estimate the friction parameter, θ, the initial conditions, $y(t_0, \theta)$, and the scale of the noise in the problem. The full Stan model is:

```
functions {
  vector sho(real t,
             vector y,
             real theta) {
    vector[2] dydt;
    dydt[1] = y[2];
    dydt[2] = -y[1] - theta * y[2];
    return dydt;
  }
}
data {
  int<lower=1> T;
  array[T] vector[2] y;
  real t0;
  array[T] real ts;
}
parameters {
  vector[2] y0;
  vector<lower=0>[2] sigma;
  real theta;
}
model {
  array[T] vector[2] mu = ode_rk45(sho, y0, t0, ts, theta);
  sigma ~ normal(0, 2.5);
  theta ~ std_normal();
  y0 ~ std_normal();
  for (t in 1:T) {
    y[t] ~ normal(mu[t], sigma);
  }
}
```

Because the solves are now a function of model parameters, the `ode_rk45` call is now made in the model block. There are half-normal priors on the measurement error scales `sigma`, and standard normal priors on `theta` and the initial state vector `y0`. The solutions to the ODE are assigned to `mu`, which is used as the location for the normal observation model.

As with other regression models, it's easy to change the noise model to something with heavier tails (e.g., Student-t distributed), correlation in the state variables (e.g., with a multivariate normal distribution), or both heavy tails and correlation in the

state variables (e.g., with a multivariate Student-t distribution).

13.5. Stiff ODEs

Stiffness is a numerical phenomena that causes some differential equation solvers difficulty, notably the Runge-Kutta RK45 solver used in the examples earlier. The phenomena is common in chemical reaction systems, which are often characterized by having multiple vastly different time-scales. The stiffness of a system can also vary between different parts of parameter space, and so a typically non-stiff system may exhibit stiffness occasionally. These sorts of difficulties can occur more frequently with loose priors or during warmup.

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 solver the `rk45` suffix is replaced with `bdf`, as in

```
ode_bdf(sho, y0, t0, ts, theta);
```

Using the stiff (`bdf`) solver on a system that is not stiff may be much slower than using the non-stiff (`rk45`) solver because each step of the stiff solver takes more time to compute. On the other hand, attempting to use the non-stiff solver for a stiff system will cause the timestep to become very small, leading the non-stiff solver taking more time overall even if each step is easier to compute than for the stiff solver.

If it is not known for sure that an ODE system is stiff, run the model with both the `rk45` and `bdf` solvers and see which is faster. If the `rk45` solver is faster, then the problem is probably non-stiff, and then it makes sense to try the `adams` solver as well. The `adams` solver uses higher order methods which can take larger timesteps than the `rk45` solver, though similar to the `bdf` solver each of these steps is more expensive to compute.

13.6. Control parameters for ODE solving

For additional control of the solves, both the stiff and non-stiff forward ODE solvers have function signatures that makes it possible to specify the `relative_tolerance`, `absolute_tolerance`, and `max_num_steps` parameters. These are the same as the regular function names but with `_tol` appended to the end. All three control arguments must be supplied with this signature (there are no defaults).

```
array[T] vector[2] y_sim = ode_bdf_tol(sho, y0, t0, ts,
                                       relative_tolerance,
```

```
                                    absolute_tolerance,
                                    max_num_steps,
                                    theta);
```

`relative_tolerance` and `absolute_tolerance` control accuracy the solver tries to achieve, and `max_num_steps` specifies the maximum number of steps the solver will take between output time points before throwing an error.

The control parameters must be data variables – they cannot 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.

For the RK45 and Cash-Karp solvers, the default values for relative and absolute tolerance are both 10^{-6} and the maximum number of steps between outputs is one million. For the BDF and Adams solvers, the relative and absolute tolerances are 10^{-10} and the maximum number of steps between outputs is one hundred million.

Discontinuous ODE system function

If there are discontinuities in the ODE system function, it is best to integrate the ODE between the discontinuities, stopping the solver at each one, and restarting it on the other side.

Nonetheless, the ODE solvers will attempt to integrate over discontinuities they encounters in the state function. The accuracy of the solution 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 zero for times $0 < t < t'$ and then positive for $t \geq t'$. In this example example, we would use code in the system such as

```
if (t < t_lag) {
  return [0, 0]';
} else {
  // ... return non-zero vector...
}
```

In general it is better to integrate up to `t_lag` in one solve and then integrate from `t_lag` onwards in another. Mathematically, the discontinuity can make the problem ill-defined and the numerical integrator may behave erratically around it.

If the location of the discontinuity cannot be controlled precisely, or there is some other rapidly change in ODE behavior, it can be useful to tell the ODE solver

to produce output in the neighborhood. This can help the ODE solver avoid indiscriminately stepping over an important feature of the solution.

Tolerance

The relative tolerance RTOL and absolute tolerance ATOL control the accuracy of the numerical solution. Specifically, when solving an ODE with unknowns $y = (y_1, \ldots, y_n)^T$, at every step the solver controls estimated local error $e = (e_1, \ldots, e_n)^T$ through its weighted root-mean-square norm (Serban and Hindmarsh (2005), Hairer, Nørsett, and Wanner (1993))

$$\sqrt{\sum_{i=1}^{n} \frac{1}{n} \frac{e_i^2}{(\text{RTOL} \times y_i + \text{ATOL})^2}} < 1$$

by reducing the stepsize when the inequality is not satisfied.

To understand the roles of the two tolerances it helps to assume y at opposite scales in the above expression: on one hand the absolute tolerance has little effect when $y_i \gg 1$, on the other the relative tolerance can not affect the norm when $y_i = 0$. Users are strongly encouraged to carefully choose tolerance values according to the ODE and its application. One can follow Brenan, Campbell, and Petzold (1995) for a rule of thumb: let m be the number of significant digits required for y, set RTOL $= 10^{-(m+1)}$, and set ATOL at which y becomes insignificant. Note that the same weighted root-mean-square norm is used to control nonlinear solver convergence in `bdf` and `adams` solvers, and the same tolerances are used to control forward sensitivity calculation. See Serban and Hindmarsh (2005) for details.

Maximum number of steps

The maximum number of steps can be used to stop a runaway simulation. This can arise in when MCMC moves to a part of parameter space very far from where a differential equation would typically be solved. In particular this can happen during warmup. With the non-stiff solver, this may happen when the sampler moves to stiff regions of parameter space, which will requires small step sizes.

13.7. Adjoint ODE solver

The adjoint ODE solver method differs mathematically from the forward ODE solvers in the way gradients of the ODE solution are obtained. The forward ODE approach augments the original ODE system with N additional states for each parameter for which gradients are needed. If there are M parameters for which sensitivities are required, then the augmented ODE system has a total of $N \cdot (M + 1)$

states. This can result in very large ODE systems through the multiplicative scaling of the computational effort needed.

In contrast, the adjoint ODE solver integrates forward in time a system of N equations to compute the ODE solution and then integrates backwards in time another system of N equations to get the sensitivities. Additionally, for M parameters there are M additional equations to integrate during the backwards solve. Because of this the adjoint sensitivity problem scales better in parameters than the forward sensitivity problem. The adjoint solver in Stan uses CVODES (the same as the `bdf` and `adams` forward sensitivity interfaces).

The solution computed in the forward integration is required during the backward integration. CVODES uses a checkpointing scheme that saves the forward solver state regularly. The number of steps between saving checkpoints is configurable in the interface. These checkpoints are then interpolated during the backward solve using one of two interpolation schemes.

The solver type (either `bdf` or `adams`) can be individually set for both the forward and backward solves.

The tolerances for each phase of the solve must be specified in the interface. Note that the absolute tolerance for the forward and backward ODE integration phase need to be set for each ODE state separately. The harmonic oscillator example call from above becomes:

```
array[T] vector[2] y_sim
    = ode_adjoint_tol_ctl(sho, y0, t0, ts,
                          relative_tolerance/9.0,                 // forwar
                          rep_vector(absolute_tolerance/9.0, 2), // forwar
                          relative_tolerance/3.0,                 // backwa
                          rep_vector(absolute_tolerance/3.0, 2), // backwa
                          relative_tolerance,                     // quadra
                          absolute_tolerance,                     // quadra
                          max_num_steps,
                          150,                                    // number
                          1,                                      // interp
                          2,                                      // solver
                          2,                                      // solver
                          theta);
```

For a detailed information on each argument please see the Stan function reference manual.

13.8. Solving a system of linear ODEs using a matrix exponential

Linear systems of ODEs can be solved using a matrix exponential. This can be considerably faster than using one of the ODE solvers.

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.

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;
  array[T] real ts;
  array[1] real theta;
}
model {
}
generated quantities {
  array[T] vector[2] y_sim;
  matrix[2, 2] A = [[ 0,  1],
```

```
                    [-1, -theta[1]]]
  for (t in 1:T) {
    y_sim[t] = matrix_exp((t - 1) * A) * y0;
  }
  // add measurement error
  for (t in 1:T) {
    y_sim[t, 1] += normal_rng(0, 0.1);
    y_sim[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_sim`. Because the ODEs are linear, we can use the `matrix_exp` function to solve the system.

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)
                    array[] real theta, // parameters
                    array[] real x_r,   // data (real)
                    array[] 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;
  array[N] real y;
}

transformed data {
  array[0] real x_r;
  array[0] int x_i;
}

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 vs. 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 double exponential 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`). If the Boost integrator is not able to reach the relative tolerance an exception is raised with a message somehing like "Exception: integrate: error estimate of integral 4.25366e-13 exceeds the given relative tolerance times norm of integral". If `integrate_1d` causes an exception in transformed parameters block or model block, the result has the same effect as assigning a $-\infty$ log probability, which causes rejection of the current proposal in MCMC samplers and adjustment of search parameters in optimization. If `integrate_1d` causes an exception in generated quantities block, the returned output from `integrate_1d` is NaN. In these cases, a bigger `relative_tolerance` value can be specified.

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, array[] real theta, array[] real x_r, array[] 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 — `a - x` or `b - x` for a lower endpoint `a` and an upper endpoint `b`. To make use of this for the beta integral, the integrand can be re-coded:

```
real beta(real x, real xc, array[] real theta, array[] real x_r, array[] 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;
}
```

In this case, as we approach the upper limit of integration $a = 1$, `xc` will take on the value of $a - x = 1 - x$. This version of the integrand will converge for much smaller values of `alpha` and `beta` than otherwise possible.

Consider another example: let's say we have a log-normal distribution that is both shifted away from zero by some amount δ, and truncated at some value b. If we were interested in calculating the expectation of a variable X distributed in this way, we would need to calculate

$$\int_a^b x f(x)\, dx = \int_\delta^b x f(x)\, dx$$

in the numerator, where $f(x)$ is the probability density function for the shifted log-normal distribution. This probability density function can be coded in Stan as:

```
real shift_lognormal_pdf(real x,
                         real mu,
                         real sigma,
                         real delta) {
  real p;

  p = (1.0 / ((x - delta) * sigma * sqrt(2 * pi()))) *
    exp(-1 * (log(x - delta) - mu)^2 / (2 * sigma^2));

  return p;
}
```

Therefore, the function that we want to integrate is:

```
real integrand(real x,
               real xc,
```

```
                array[] real theta,
                array[] real x_r,
                array[] int x_i) {
  real numerator;
  real p;

  real mu = theta[1];
  real sigma = theta[2];
  real delta = theta[3];
  real b = theta[4];

  p = shift_lognormal_pdf(x, mu, sigma, delta);

  numerator = x * p;

  return numerator;
}
```

What happens here is that, given that the log-normal distribution is shifted by δ, when we then try to integrate the numerator, our `x` starts at values just above `delta`. This, in turn, causes the `x - delta` term to be near zero, leading to a breakdown.

We can use `xc`, and define the `integrand` as:

```
real integrand(real x,
                real xc,
                array[] real theta,
                array[] real x_r,
                array[] int x_i) {
  real numerator;
  real p;

  real mu = theta[1];
  real sigma = theta[2];
  real delta = theta[3];
  real b = theta[4];

  if (x < delta + 1) {
    p = shift_lognormal_pdf(xc, mu, sigma, delta);
  } else {
```

```
    p = shift_lognormal_pdf(x, mu, sigma, delta);
  }

  numerator = x * p;

  return numerator;
}
```

Why does this work? When our values of `x` are less than `delta + 1` (so, when they're near `delta`, given that our lower bound of integration is equal to δ), we pass `xc` as an argument to our `shift_lognormal_pdf` function. This way, instead of dealing with `x - delta` in `shift_lognormal_pdf`, we are working with `xc - delta` which is equal to `delta - x - delta`, as `delta` is the lower endpoint in that case. The `delta` terms cancel out, and we are left with a high-precision version of `x`. We don't encounter the same problem at the upper limit b so we don't adjust the code for that case.

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 a high precision version of `a - x`, `x`, or `b - x`, depending on the value of `x` and the endpoints.

15. Complex Numbers

Stan supports complex scalars, matrices, and vectors as well as real-based ones.

15.1. Working with complex numbers

This section describes the complex scalar type, including how to build complex numbers, assign them, and use them in arrays and functions.

Constructing and accessing complex numbers

Complex numbers can be constructed using imaginary literals. For example,

```
complex z = -1.1 + 2.3i;
```

produces the complex number $-1.1 + 2.3i$. This only works if the real and imaginary components are literal numerals. To construct a complex number out of arbitrary real variables, the `to_complex()` function may be used. For example, the following code will work if `x` and `y` are parameters, transformed data, or local variables in a function or model block.

```
real x = // ...
real y = // ...
complex z = to_complex(x, y);
```

The real and imaginary parts of the complex number can be accessed with getters as follows.

```
real x = get_real(z);  // x = -1.1
real y = get_imag(z);  // y = 2.3
```

Complex numbers can be compared using equality (or inequality), but not with greater than or less than operators. For example, after running the code above, the following code snippet will print "hello".

```
complex a = 3.2 + 2i;
complex b = to_complex(3.2, 2);
if (a == b) print("hello");
```

Complex assignment and promotion

Integer- or real-valued expressions may be assigned to complex numbers, with the corresponding imaginary component set to zero.

```
complex z1 = 3;  // int promoted to 3 + 0i
complex z2 = 3.2;  // real promoted to 3.2 + 0.i
```

Complex arrays

Arrays of complex numbers work as usual and allow the usual curly bracket constructors.

```
complex z1;  complex z2;  complex z3;
// ...
array[3] complex zs = { z1, z2, z3 };
for (z in zs) {
  print(z);
}
```

Complex arrays allow assignment into their elements, with integer or real assigned values being promoted to complex.

Complex functions

All of the standard complex functions are available, including natural logarithm `log(z)`, natural exponentiation `exp(z)`, and powers `pow(z1, z2)`, as well as all of the trig and hyperbolic trigonometric functions and their inverse, such as `sin(z)`, `acos(z)`, `tanh(z)` and `asinh(z)`.

Promotion also works for complex-valued function arguments, which may be passed integer or real values, which will be promoted before the function is evaluated. For example, the following user-defined complex function will accept integer, real, or complex arguments.

```
complex times_i(complex z) {
  complex i = to_complex(0, 1);
  return i * z;
}
```

For instance, `times_i(1)` evaluates to the imaginary base i, as does `times_i(1.0)`.

15.2. Complex random variables

The simplest way to model a distribution over a complex random number $z = x + yi$ is to consider its real part x and imaginary part y to have a bivariate normal

distribution. For example, a complex prior can be expressed as follows.

```
complex z;
vector[2] mu;
cholesky_cov[2] L_Sigma;
// ...
[get_real(z), get_imag(z)]' ~ multi_normal_cholesky(mu, L_Sigma);
```

For example, if z is data, this can be used to estimate mu and the covariance Cholesky factor L_Sigma. Alternatively, if z is a parameter, mu and L_Sigma may constants defining a prior or further parameters defining a hierarchical model.

15.3. Complex matrices and vectors

Stan supports complex matrices, vectors, and row vectors. Variables of these types are declared with sizes in the same way as their real-based counterparts.

```
complex_vector[3] v;
complex_row_vector[2] rv;
complex_matrix[3, 2] m;
```

We can construct vectors and matrices using brackets in the same way as for real-valued vectors and matrices. For example, given the declaration of rv above, we could assign it to a constructed row vector.

```
rv =  [2 + 3i, 1.9 - 2.3i];
```

Complex matrices and vectors support all of the standard arithetmic operations including negation, addition, subtraction, and multiplication (division involves a solve, and isn't a simple arithmetic operation for matrices). They also support transposition.

Furthermore, it is possible to convert back and forth between arrays and matrices using the to_array functions.

15.4. Complex linear regression

Complex valued linear regression with complex predictors and regression coefficients looks just like standard regression. For example, if we take x to be predictors, y to be an array of outcomes. For example, consider the following complete Stan program for an intercept and slope.

```
data {
  int<lower=0> N;
```

```
  complex_vector[N] x;
  complex_vector[N] y;
}
parameters {
  complex alpha;
  complex beta;
}
model {
  complex_vector[N] eps = y - (alpha + beta * x);
  eps ~  // ...error distribution...
}
```

The question remains of how to fill in the error distribution and there are several alternatives. We consider only two simple alternatives, and do not consider penalizing the absolute value of the error.

Independent real and imaginary error

The simplest approach to error in complex regression is to give the real and imaginary parts of `eps_n` independent independent normal distributions, as follows.

```
parameters {
  // ...
  vector[2] sigma;
}
// ...
model {
  // ...
  get_real(eps) ~ normal(0, sigma[1]);
  get_imag(eps) ~ normal(0, sigma[2]);
  sigma ~ //...hyperprior...
}
```

A new error scale vector `sigma` is introduced, and it should itself get a prior based on the expected scale of error for the problem.

Dependent complex error

The next simplest approach is to treat the real and imaginary parts of the complex number as having a multivariate normal prior. This can be done by adding a parameter for correlation to the above, or just working with a multivariate covariance matrix, as we do here.

```
parameters {
  cholesky_factor_corr[2] L_Omega;  // correlation matrix
  vector[2] sigma;                  // real, imag error scales
  // ...
}
// ...
model {
  array[N] vector[2] eps_arr;
  for (n in 1:N) {
    eps_arr[n] = { to_real(eps[n]), to_imag(eps[n]) };
  }
  eps_arr ~ multi_normal_cholesky([0, 0]',
                                  diag_pre_multiply(sigma, L_Omega));
  L_Omega ~ lkj_cholesky(4);  // shrink toward diagonal correlation
  sigma ~ // ... hyperprior ...
}
```

Here, the real and imaginary components of the error get a joint distribution with correlation and independent scales. The error gets a multivariate normal distribution with zero mean and a Cholesky factor representation of covariance, consisting of a scale vector `sigma` and a Cholesky factor or a correlation matrix, `L_Omega`. The prior on the correlations is concentrated loosely around diagonal covariance, and the prior on the scales is left open. In order to vectorize the call to `multi_normal_cholesky`, the vector of complex numbers needs to be converted to an array of size 2 vectors.

16. Differential-Algebraic Equations

Stan support solving systems of differential-algebraic equations (DAEs) of index 1 (Serban et al. 2021). The solver adaptively refines the solutions in order to satisfy given tolerances.

One can think a differential-algebraic system of equations as ODEs with additional algebraic constraints applied to some of the variables. In such a system, the variable derivatives may not be expressed explicitly with a right-hand-side as in ODEs, but implicitly constrained.

Similar to ODE solvers, the DAE solvers must not only provide the solution to the DAE itself, but also the gradient of the DAE solution with respect to parameters (the sensitivities). Stan's DAE solver uses the *forward sensitivity* calculation to expand the base DAE system with additional DAE equations for the gradients of the solution. For each parameter, an additional full set of N sensitivity states are added meaning that the full DAE solved has $N + N \cdot M$ states.

Two interfaces are provided for the forward sensitivity solver: one with default tolerances and default max number of steps, and one that allows these controls to be modified. Choosing tolerances is important for making any of the solvers work well – the defaults will not work everywhere. The tolerances should be chosen primarily with consideration to the scales of the solutions, the accuracy needed for the solutions, and how the solutions are used in the model. The same principles in the control parameters section apply here.

Internally Stan's DAE solver uses a variable-step, variable-order, backward-differentiation formula implementation (Cohen and Hindmarsh 1996; Serban and Hindmarsh 2005).

16.1. Notation

A DAE is defined by a set of expressions for the *residuals* of differential equations and algebraic equations $r(y', y, t, \theta)$, and *consistent* initial conditions $y(t_0, \theta) = y_0, y'(t_0, \theta) = y'_0$. The DAE is define by residual function as $r(y', y, t, \theta) = 0$. The θ dependence is included in the notation to highlight that the solution $y(t)$ is a function of any parameters used in the computation.

16.2. Example: chemical kinetics

As an example of a system of DAEs, consider following chemical kinetics problem(Robertson 1966). The nondimensionalized DAE consists of two differential equations and one algebraic constraint. The differential equations describe the reactions from reactants y_1 and y_2 to the product y_3, and the algebraic equation describes the mass conservation. (Serban and Hindmarsh 2021).

$$\begin{aligned} \frac{dy_1}{dt} + \alpha y_1 - \beta y_2 y_3 &= 0 \\ \frac{dy_2}{dt} - \alpha y_1 + \beta y_2 y_3 + \gamma y_2^2 &= 0 \\ y_1 + y_2 + y_3 - 1.0 &= 0 \end{aligned}$$

The state equations implicitly defines the state $(y_1(t), y_2(t), y_3(t))$ at future times as a function of an initial state and the system parameters, in this example the reaction rate coefficients (α, β, γ).

Unlike solving ODEs, solving DAEs requires a *consistent* initial condition. That is, one must specify both $y(t_0)$ and $y'(t_0)$ so that residual function becomes zero at initial time t_0

$$r(y'(t_0), y(t_0), t_0) = 0$$

16.3. Index of DAEs

The index along a DAE solution $y(t)$ is the minimum number of differentiations of some of the components of the system required to solve for y' uniquely in terms of y and t, so that the DAE is converted into an ODE for y. Thus an ODE system is of index 0. The above chemical kinetics DAE is of index 1, as we can perform differentiation of the third equation followed by introducing the first two equations in order to obtain the ODE for y_3.

Most DAE solvers, including the one in Stan, support only index-1 DAEs. For a high index DAE problem the user must first convert it to a lower index system. This often can be done by carrying out differentiations analytically (Ascher and Petzold 1998).

16.4. Coding the DAE system function

The first step in coding an DAE system in Stan is defining the DAE residual function. The system functions require a specific signature so that the solvers know how to use them properly.

The first argument to the residual function is time, passed as a `real`; the second argument to the residual function is the system state y, passed as a `vector`, the third argument to the residual function is the state derivative y', also passed as a `vector`. The residual function's return value is a `vector` of the same size as state and state derivatives. Additional arguments can be included in the residual function to pass other information into the solve (these will be passed through the function that starts the DAE solution). These argument can be parameters (in our example, the reaction rate coefficient α, β, and γ), data, or any quantities that are needed to define the DAE.

The above reaction be coded using the following function in Stan (see the user-defined functions chapter for more information on coding user-defined functions).

```
 vector chem(real t, vector yy, vector yp,
                 real alpha, real beta, real gamma) {
    vector[3] res;
    res[1] = yp[1] + alpha * yy[1] - beta * yy[2] * yy[3];
    res[2] = yp[2] - alpha * yy[1] + beta * yy[2] * yy[3] + gamma * yy[2]
    res[3] = yy[1] + yy[2] + yy[3] - 1.0;
    return res;
  }
}
```

The function takes in a time `t` (a `real`), the system state `yy` (a `vector`), state derivative `yp` (a `vector`), as well as parameter `alpha` (a `real`), `beta` (a `real`), and `gamma` (a `real`). The function returns a `vector` of the residuals at time `t`. The DAE coded here does not explicitly depend on `t`, however one still needs to specify `t` as an argument.

Strict signature

The types in the DAE residual function are strict. The first argument is the time passed as a `real`, the second argument is the state passed as a `vector`, the third argument is the state derivative passed as a `vector`, and the return type is a `vector`. A model that does not have this signature will fail to compile. The fourth argument onwards can be any type, granted all the argument types match the types of the respective arguments in the solver call.

All of these are possible DAE signatures:

```
vector my_dae1(real t, vector y, vector yp, real a0);
vector my_dae2(real t, vector y, vector yp, array[] int a0, vector a1);
vector my_dae3(real t, vector y, vector yp, matrix a0, array[] real a1, rc
```

but these are not allowed:

```
vector my_dae1(real t, array[] real y, vector yp);
// Second argument is not a vector
array[] real my_dae2(real t, vector y, vector yp);
// Return type is not a vector
vector my_dae3(real t, vector y);
// First argument is not a real and missing the third argument
```

16.5. Solving DAEs

Stan provides a `dae` function for solving DAEs, so that the above chemical reaction equation can be solved in the following code.

```
data {
  int N;
  vector[3] yy0;
  vector[3] yp0;
  real t0;
  real alpha;
  real beta;
  array[N] real ts;
  array[N] vector[3] y;
}
parameters {
  real gamma;
}
transformed parameters {
  vector[3] y_hat[N] = dae(chem, yy0, yp0, t0, ts, alpha, beta, gamma);
}
```

Since `gamma` is a parameter, the DAE solver is called in the transformed parameters block.

16.6. Control parameters for DAE solving

Using `dae_tol` one can specify the `relative_tolerance`, `absolute_tolerance`, and `max_num_steps` parameters in order to control the DAE solution.

```
vector[3] y_hat[N] = dae_tol(chem, yy0, yp0, t0, ts,
                             relative_tolerance,
                             absolute_tolerance,
```

```
                                max_num_steps,
                                alpha, beta, gamma);
```

`relative_tolerance` and `absolute_tolerance` control accuracy the solver tries to achieve, and `max_num_steps` specifies the maximum number of steps the solver will take between output time points before throwing an error.

The control parameters must be data variables – they cannot 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.

The default value of relative and absolute tolerances are 10^{-10} and the maximum number of steps between outputs is one hundred million. We suggest the user choose the control parameters according to the problem in hand, and resort to the defaults only when no knowledge of the DAE system or the physics it models is available.

Maximum number of steps

The maximum number of steps can be used to stop a runaway simulation. This can arise in when MCMC moves to a part of parameter space very far from where a differential equation would typically be solved. In particular this can happen during warmup. With the non-stiff solver, this may happen when the sampler moves to stiff regions of parameter space, which will requires small step sizes.

17. Survival Models

Survival models apply to animals and plants as well as inanimate objects such as machine parts or electrical components. Survival models arise when there is an event of interest for a group of subjects, machine component, or other item that is

- certain to occur after some amount of time,
- but only measured for a fixed period of time, during which the event may not have occurred for all subjects.

For example, one might wish to estimate the the distribution of time to failure for solid state drives in a data center, but only measure drives for a two year period, after which some number will have failed and some will still be in service.

Survival models are often used comparatively, such as comparing time to death of patients diagnosed with stage one liver cancer under a new treatment and a standard treatment (pure controls are not allowed when there is an effective existing treatment for a serious condition). During a two year trial, some patients will die and others will survive.

Survival models may involve covariates, such as the factory at which a component is manufactured, the day on which it is manufactured, and the amount of usage it gets. A clinical trial might be adjusted for the sex and age of a cancer patient or the hospital at which treatment is received.

Survival models come in two main flavors, parametric and semi-parametric. In a parametric model, the survival time of a subject is modeled explicitly using a parametric probability distribution. There is a great deal of flexibility in how the parametric probability distribution is constructed. The sections below consider exponential and Weibull distributed survival times.

Rather than explicitly modeling a parametric survival probability, semi-parametric survival models instead model the relative effect on survival of covariates. The final sections of this chapter consider the proportional hazards survival model.

17.1. Exponential survival model

The exponential distribution is commonly used in survival models where there is a constant risk of failure that does not go up the longer a subject survives. This is because the exponential distribution is memoryless in sense that if $T \sim$

exponential(λ) for some rate $\lambda > 0$, then

$$\Pr[T > t] = \Pr[T > t + t' \mid T > t'].$$

If component survival times are distributed exponentially, it means the distribution of time to failure is the same no matter how long the item has already survived. This can be a reasonable assumption for electronic components, but is not a reasonable model for animal survival.

The exponential survival model has a single parameter for the rate, which assumes all subjects have the same distribution of failure time (this assumption is relaxed in the next section by introducing per-subject covariates). With the rate parameterization, the expected survival time for a component with survival time represented as the random variable T is

$$\mathbb{E}[T \mid \lambda] = \frac{1}{\lambda}.$$

The exponential distribution is sometimes parameterized in terms of a scale (i.e., inverse rate) $\beta = 1/\lambda$.

The data for a survival model consists of two components. First, there is a vector $t \in (0, \infty)^N$ of N observed failure times. Second, there is a censoring time t^{cens} such that failure times greater than t^{cens} are not observed. The censoring time assumption imposes a constraint which requires $t_n < t^{\text{cens}}$ for all $n \in 1{:}N$. For the censored subjects, the only thing required in the model is their total count, N^{cens} (their covariates are also required for models with covariates).

The model for the observed failure times is exponential, so that for $n \in 1{:}N$,

$$t_n \sim \text{exponential}(\lambda).$$

The model for the censored failure times is also exponential. All that is known of a censored item is that its failure time is greater than the censoring time, so each censored item contributes a factor to the likelihood of

$$\Pr[T > t^{\text{cens}}] = 1 - F_T(t^{\text{cens}}),$$

where F_T is the cumulative distribution function (cdf) of survival time T ($F_X(x) = \Pr[X \leq x]$ is standard notation for the cdf of a random variable X). The function $1 - F_T(t)$ is the complementary cumulative distribution function (ccdf), and it is

used directly to define the likelihood

$$\begin{aligned} p(t, t^{\text{cens}}, N^{\text{cens}} \mid \lambda) &= \prod_{n=1}^{N} \text{exponential}(t_n \mid \lambda) \cdot \prod_{n=1}^{N^{\text{cens}}} \text{exponentialCCDF}(t^{\text{cens}} \mid \lambda) \\ &= \prod_{n=1}^{N} \text{exponential}(t_n \mid \lambda) \cdot \text{exponentialCCDF}(t^{\text{cens}} \mid \lambda)^{N^{\text{cens}}}. \end{aligned}$$

On the log scale, that's

$$\begin{aligned} \log p(t, t^{\text{cens}}, N^{\text{cens}} \mid \lambda) &= \sum_{n=1}^{N} \log \text{exponential}(t_n \mid \lambda) \\ &\quad + N^{\text{cens}} \cdot \log \text{exponentialCCDF}(t^{\text{cens}} \mid \lambda). \end{aligned}$$

The model can be completed with a standard lognormal prior on λ,

$$\lambda \sim \text{lognormal}(0, 1),$$

which is reasonable if failure times are in the range of 0.1 to 10 time units, because that's roughly the 95% central interval for a variable distributed lognormal$(0, 1)$. In general, the range of the prior (and likelihood!) should be adjusted with prior knowledge of expected failure times.

Stan program

The data for a simple survival analysis without covariates can be coded as follows.

```
data {
  int<lower=0> N;
  vector[N] t;
  int<lower=0> N_cens;
  real<lower=0> t_cens;
}
```

In this program, `N` is the number of uncensored observations and `t` contains the times of the uncensored observations. There are a further `N_cens` items that are right censored at time `t_cens`. Right censoring means that if the time to failure is greater than

`t_cens`, it is only observed that the part survived until time `t_cens`. In the case where there are no covariates, the model only needs the number of censored items because they all share the same censoring time.

There is a single rate parameter, the inverse of which is the expected time to failure.

```
parameters {
  real<lower=0> lambda;
}
```

The exponential survival model and the prior are coded directly using vectorized distribution and ccdf statements. This both simplifies the code and makes it more computationally efficient by sharing computation across instances.

```
model {
  t ~ exponential(lambda);
  target += N_cens * exponential_lccdf(t_cens | lambda);

  lambda ~ lognormal(0, 1);
}
```

The likelihood for rate `lambda` is just the density of exponential distribution for observed failure time. The Stan code is vectorized, modeling each entry of the vector `t` as a having an exponential distribution with rate `lambda`. This data model could have been written as

```
for (n in 1:N) {
  t[n] ~ exponential(lambda);
}
```

The log likelihood contribution given censored items is the number of censored items times the log complementary cumulative distribution function (lccdf) at the censoring time of the exponential distribution with rate `lambda`. The log likelihood terms arising from the censored events could have been added to the target log density one at a time,

```
for (n in 1:N)
  target += exponential_lccdf(t_cens | lambda);
```

to define the same log density, but it is much more efficient computationally to multiply by a constant than do a handful of sequential additions.

17.2. Weibull survival model

The Weibull distribution is a popular alternative to the exponential distribution in cases where there is a decreasing probability of survival as a subject gets older. The Weibull distribution models this by generalizing the exponential distribution to include a power-law trend.

The Weibull distribution is parameterized by a shape $\alpha > 0$ and scale $\sigma > 0$. For an outcome $t \geq 0$, the Weibull distribution's probability density function is

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

In contrast, recall that the exponential distribution can be expressed using a rate (inverse scale) parameter $\beta > 0$ with probability density function

$$\text{exponential}(t \mid \beta) = \beta \cdot \exp(-\beta \cdot t).$$

When $\alpha = 1$, the Weibull distribution reduces to an exponential distribution,

$$\text{Weibull}(t \mid 1, \sigma) = \text{exponential}\left(t \,\middle|\, \frac{1}{\sigma}\right).$$

In other words, the Weibull is a continuous expansion of the exponential distribution.

If $T \sim \text{Weibull}(\alpha, \sigma)$, then the expected survival time is

$$\mathbb{E}[T] = \sigma \cdot \Gamma\left(1 + \frac{1}{\alpha}\right),$$

where the Γ function is the continuous completion of the factorial function (i.e., $\Gamma(1 + n) = n!$ for $n \in \mathbb{N}$). As $\alpha \to 0$ for a fixed σ or as $\sigma \to \infty$ for a fixed α, the expected survival time goes to infinity.

There are three regimes of the Weibull distribution.

- $\alpha < 1$. A subject is more likely to fail early. When $\alpha < 1$, the Weibull density approaches infinity as $t \to 0$.
- $\alpha = 1$. The Weibull distribution reduces to the exponential distribution, with a constant rate of failure over time. When $\alpha = 1$, the Weibull distribution approaches σ as $t \to 0$.
- $\alpha > 1$. Subjects are less likely to fail early. When $\alpha < 1$, the Weibull density approaches zero as $t \to 0$.

With $\alpha \leq 1$, the mode is zero ($t = 0$), whereas with $\alpha > 1$, the mode is nonzero ($t > 0$).

Stan program

With Stan, one can just swap the exponential distribution for the Weibull distribution with the appropriate parameters and the model remains essentially the same. Recall the exponential model's parameters and model block.

```
parameters {
  real<lower=0> beta;
}
model {
  t ~ exponential(beta);
  target += N_cens * exponential_lccdf(t_cens | beta);

  beta ~ lognormal(0, 1);
}
```

The Stan program for the Weibull model just swaps in the Weibull distribution and complementary cumulative distribution function with shape (`alpha`) and scale (`sigma`) parameters.

```
parameters {
  real<lower=0> alpha;
  real<lower=0> sigma;
}
model {
  t ~ weibull(alpha, sigma);
  target += N_cens * weibull_lccdf(t_cens | alpha, sigma);

  alpha ~ lognormal(0, 1);
  sigma ~ lognormal(0, 1);
}
```

As usual, if more is known about expected survival times, `alpha` and `sigma` should be given more informative priors.

17.3. Survival with covariates

Suppose that for each of $n \in 1{:}N$ items observed, both censored and uncensored, there is a covariate (row) vector $x_n \in \mathbb{R}^K$. For example, a clinical trial may include the age (or a one-hot encoding of an age group) and the sex of a participant; an electronic component might include a one-hot encoding of the factory at which it was manufactured and a covariate for the load under which it has been run.

Survival with covariates replaces what is essentially a simple regression with only an intercept λ with a generalized linear model with a log link, where the rate for item n is

$$\lambda_n = \exp(x_n \cdot \beta),$$

where $\beta \in \mathbb{R}^K$ is a K-vector of regression coefficients. Thus

$$t_n \sim \text{exponential}(\lambda_n).$$

The censored items have probability

$$\Pr[n\text{-th censored}] = \text{exponentialCCDF}(t^{\text{cens}} \mid x_n^{\text{cens}} \cdot \beta).$$

The covariates form an $N \times K$ data matrix, $x \in \mathbb{R}^{N \times K}$. An intercept can be introduced by adding a column of 1 values to x.

A Stan program for the exponential survival model with covariates is as follows. It relies on the fact that the order of failure times (`t` and `t_cens`) corresponds to the ordering of items in the covariate matrices (`x` and `x_cens`).

```
data {
  int<lower=0> N;
  vector[N] t;
  int<lower=0> N_cens;
  real<lower=0> t_cens;
  int<lower=0> K;
  matrix[N, K] x;
  matrix[N_cens, K] x_cens;
}
parameters {
  vector[K] gamma;
}
model {
  gamma ~ normal(0, 2);

  t ~ exponential(exp(x * gamma));
  target += exponential_lccdf(t_cens | exp(x_cens * gamma));
}
```

Both the distribution statement for uncensored times and the log density increment statement for censored times are vectorized, one in terms of the exponential distribution and one in terms of the log complementary cumulative distribution function.

17.4. Hazard and survival functions

Suppose T is a random variable representing a survival time, with a smooth cumulative distribution function

$$F_T(t) = \Pr[T \leq t],$$

so that its probability density function is

$$p_T(t) = \frac{\mathrm{d}}{\mathrm{d}t} F_T(t).$$

The *survival function* $S(t)$ is the probability of surviving until at least time t, which is just the complementary cumulative distribution function (ccdf) of the survival random variable T,

$$S(t) = 1 - F_T(t).$$

The survival function appeared in the Stan model in the previous section as the likelihood for items that did not fail during the period of the experiment (i.e., the censored failure times for the items that survived through the trial period).

The *hazard function* $h(t)$ is the instantaneous risk of not surviving past time t assuming survival until time t, which is given by

$$h(t) = \frac{p_T(t)}{S(t)} = \frac{p_T(t)}{1 - F_T(t)}.$$

The *cumulative hazard function* $H(t)$ is defined to be the accumulated hazard over time,

$$H(t) = \int_0^t h(u)\,\mathrm{d}u.$$

The hazard function and survival function are related through the differential

equation

$$\begin{aligned} h(t) &= -\frac{\mathrm{d}}{\mathrm{d}t} \log S(t). \\ &= -\frac{1}{S(t)} \frac{\mathrm{d}}{\mathrm{d}t} S(t) \\ &= \frac{1}{S(t)} \frac{\mathrm{d}}{\mathrm{d}t} - (1 - F_Y(t)) \\ &= \frac{1}{S(t)} \frac{\mathrm{d}}{\mathrm{d}t} (F_Y(t) - 1) \\ &= \frac{1}{S(t)} \frac{\mathrm{d}}{\mathrm{d}t} F_Y(t) \\ &= \frac{p_T(t)}{S(t)}. \end{aligned}$$

If $T \sim \text{exponential}(\beta)$ has an exponential distribution, then its hazard function is constant,

$$\begin{aligned} h(t \mid \beta) &= \frac{p_T(t \mid \beta)}{S(t \mid \beta)} \\ &= \frac{\text{exponential}(t \mid \beta)}{1 - \text{exponentialCCDF}(t \mid \beta)} \\ &= \frac{\beta \cdot \exp(-\beta \cdot t)}{1 - (1 - \exp(-\beta \cdot t))} \\ &= \frac{\beta \cdot \exp(-\beta \cdot t)}{\exp(-\beta \cdot t)} \\ &= \beta. \end{aligned}$$

The exponential distribution is the only distribution of survival times with a constant hazard function.

If $T \sim \text{Weibull}(\alpha, \sigma)$, then its hazard function is

$$\begin{aligned} h(t \mid \alpha, \sigma) &= \frac{p_T(t \mid \alpha, \sigma)}{S(t \mid \alpha, \sigma)} \\ &= \frac{\text{Weibull}(t \mid \alpha, \sigma}{1 - \text{WeibullCCDF}(t \mid \alpha, \sigma)} \\ &= \frac{\frac{\alpha}{\sigma} \cdot \left(\frac{t}{\sigma}\right)^{\alpha-1} \cdot \exp\left(-\left(\frac{t}{\sigma}\right)^{\alpha}\right)}{1 - \left(1 - \exp\left(-\left(\frac{t}{\sigma}\right)^{\alpha}\right)\right)} \\ &= \frac{\alpha}{\sigma} \cdot \left(\frac{t}{\sigma}\right)^{\alpha-1}. \end{aligned}$$

If $\alpha = 1$ the hazard is constant over time (which also follows from the fact that the Weibull distribution reduces to the exponential distribution when $\alpha = 1$). When $\alpha > 1$, the hazard grows as time passes, whereas when $\alpha < 1$, it decreases as time passes.

17.5. Proportional hazards model

The exponential model is parametric in that is specifies an explicit parametric form for the distribution of survival times. Cox (1972) introduced a semi-parametric survival model specified directly in terms of a hazard function $h(t)$ rather than in terms of a distribution over survival times. Cox's model is semi-parametric in that it does not model the full hazard function, instead modeling only the proportional differences in hazards among subjects.

Let $x_n \in \mathbb{R}^K$ be a (row) vector of covariates for subject n so that the full covariate data matrix is $x \in \mathbb{R}^{N \times K}$. In Cox's model, the hazard function for subject n is defined conditionally in terms of their covariates x_n and the parameter vector $\gamma \in \mathbb{R}^K$ as

$$h(t \mid x_n, \beta) = h_0(t) \cdot \exp(x_n \cdot \gamma),$$

where $h_0(t)$ is a shared baseline hazard function and $x_n \cdot \gamma = \sum_{k=1}^{K} x_{n,k} \cdot \beta_k$ is a row vector-vector product.

In the semi-parametric, proportional hazards model, the baseline hazard function $h_0(t)$ is not modeled. This is why it is called "semi-parametric." Only the factor $\exp(x_n \cdot \gamma)$, which determines how individual n varies by a proportion from the baseline hazard, is modeled. This is why it's called "proportional hazards."

Cox's proportional hazards model is not fully generative. There is no way to generate the times of failure because the baseline hazard function $h_0(t)$ is unmodeled; if the baseline hazard were known, failure times could be generated. Cox's proportional hazards model is generative for the ordering of failures conditional on a number of censored items. Proportional hazard models may also include parametric or non-parametric model for the baseline hazard function[1].

Partial likelihood function

Cox's proportional specification of the hazard function is insufficient to generate random variates because the baseline hazard function $h_0(t)$ is unknown. On the other hand, the proportional specification is sufficient to generate a partial likelihood that accounts for the order of the survival times.

The hazard function $h(t \mid x_n, \beta) = h_0(t) \cdot \exp(x_n \cdot \beta)$ for subject n represents the instantaneous probability that subject n fails at time t given that it has survived until time t. The probability that subject n is the first to fail among N subjects is thus proportional to subject n's hazard function,

$$\Pr[n \text{ first to fail at time } t] \propto h(t \mid x_n, \beta).$$

Normalizing yields

$$\begin{aligned}
\Pr[n \text{ first to fail at time } t] &= \frac{h(t \mid x_n, \beta)}{\sum_{n'=1}^{N} h(t \mid x_{n'}, \beta)} \\
&= \frac{h_0(t) \cdot \exp(x_n \cdot \beta)}{\sum_{n'=1}^{N} h_0(t) \cdot \exp(x_{n'} \cdot \beta)} \\
&= \frac{\exp(x_n \cdot \beta)}{\sum_{n'=1}^{N} \exp(x_{n'} \cdot \beta)}.
\end{aligned}$$

Suppose there are N subjects with strictly *ordered* survival times $t_1 < t_2 < \cdots < t_N$ and covariate (row) vectors $x_1, \ldots, x_N$. Let t^{cens} be the (right) censoring time and let N^{obs} be the largest value of n such that $t_n \leq t^{\text{cens}}$. This means N^{obs} is the number of subjects whose failure time was observed. The ordering is for convenient indexing and does not cause any loss of generality—survival times can simply be sorted into the necessary order.

With failure times sorted in decreasing order, the partial likelihood for each observed

[1] Cox mentioned in his seminal paper that modeling the baseline hazard function would improve statistical efficiency, but he did not do it for computational reasons.

subject $n \in 1{:}N^{\text{obs}}$ can be expressed as

$$\Pr[n \text{ first to fail among } n, n+1, \ldots N] = \frac{\exp(x_n \cdot \beta)}{\sum_{n'=n}^{N} \exp(x_{n'} \cdot \beta)}.$$

The group of items for comparison and hence the summation is over all items, including those with observed and censored failure times.

The partial likelihood, defined in this form by Breslow (1975), is just the product of the partial likelihoods for the observed subjects (i.e., excluding subjects whose failure time is censored).

$$\Pr[\text{observed failures ordered } 1, \ldots, N^{\text{obs}} | x, \beta] = \prod_{n=1}^{N^{\text{obs}}} \frac{\exp(x_n \cdot \beta)}{\sum_{n'=n}^{N} \exp(x_{n'} \cdot \beta)}.$$

On the log scale,

$$\begin{aligned} \log \Pr[\text{obs. fail ordered } 1, \ldots, N^{\text{obs}} | x, \beta] &= \sum_{n=1}^{N^{\text{obs}}} \log \left(\frac{\exp(x_n \cdot \beta)}{\sum_{n'=n}^{N} \exp(x_{n'} \cdot \beta)} \right) \\ &= x_n \cdot \beta - \log \sum_{n'=n}^{N} \exp(x_{n'} \cdot \beta) \\ &= x_n \cdot \beta - \text{logSumExp}_{n'=n}^{N} \, x_{n'} \cdot \beta, \end{aligned}$$

where

$$\text{logSumExp}_{n=a}^{b} \, x_n = \log \sum_{n=a}^{b} \exp(x_n)$$

is implemented so as to preserve numerical precision.

This likelihood follows the same approach to ranking as that developed by Plackett (1975) for estimating the probability of the order of the first few finishers in a horse race.

A simple normal prior on the components of β completes the model,

$$\beta \sim \text{normal}(0, 2).$$

This should be scaled based on knowledge of the predictors.

Stan program

To simplify the Stan program, the survival times for uncensored events are sorted into decreasing order (unlike in the mathematical presentation, where they were sorted into ascending order). The covariates for censored and uncensored observations are separated into two matrices.

```
data {
  int<lower=0> K;          // num covariates

  int<lower=0> N;          // num uncensored obs
  vector[N] t;             // event time (non-strict decreasing)
  matrix[N, K] x;          // covariates for uncensored obs

  int N_c;                 // num censored obs
  real<lower=t[N]> t_c;    // censoring time
  matrix[N_c, K] x_c;      // covariates for censored obs
}
```

The parameters are just the coefficients.

```
parameters {
  vector[K] beta;          // slopes (no intercept)
}
```

The prior is a simple independent centered normal distribution on each element of the parameter vector, which is vectorized in the Stan code.

```
model {
  beta ~ normal(0, 2);
  ...
```

The log likelihood is implemented so as to minimize duplicated effort. The first order of business is to calculate the linear predictors, which is done separately for the subjects whose event time is observed and those for which the event time is censored.

```
  vector[N] log_theta = x * beta;
  vector[N_c] log_theta_c = x_c * beta;
```

These vectors are computed using efficient matrix-vector multiplies. The log of exponential values of the censored covariates times the coefficients is reused in the denominator of each factor, which on the log scale, starts with the log sum of exponentials of the censored items' linear predictors.

```
  real log_denom = log_sum_exp(log_theta_c);
```

Then, for each observed survival time, going backwards from the latest to the earliest, the denominator can be incremented (which turns into a log sum of exponentials on the log scale), and then the target is updated with its likelihood contribution.

```
  for (n in 1:N) {
    log_denom = log_sum_exp(log_denom, log_theta[n]);
    target += log_theta[n] - log_denom;   // log likelihood
  }
```

The running log sum of exponentials is why the list is iterated in reverse order of survival times. It allows the log denominator to be accumulated one term at a time. The condition that the survival times are sorted into decreasing order is not checked. It could be checked very easily in the transformed data block by adding the following code.

```
transformed data {
  for (n in 2:N) {
    if (!(t[n] < t[n - 1])) {
      reject("times must be strictly decreasing, but found"
             "!(t[", n, "] < t[, ", (n - 1), "])");
    }
  }
}
```

Stan model for tied survival times

Technically, for continuous survival times, the probability of two survival times being identical will be zero. Nevertheless, real data sets often round survival times, for instance to the nearest day or week in a multi-year clinical trial. The technically "correct" thing to do in the face of unknown survival times in a range would be to treat their order as unknown and infer it. But considering all $N!$ permutations for a set of N subjects with tied survival times is not tractable. As an alternative, Efron (1977) introduced an approximate partial likelihood with better properties than a random permutation while not being quite as good as considering all permutations. Efron's model averages the contributions as if they truly did occur simultaneously.

In the interest of completeness, here is the Stan code for an implementation of Efron's estimator. It uses two user-defined functions. The first calculates how many different survival times occur in the data.

```
functions {
  int num_unique_starts(vector t) {
    if (size(t) == 0) return 0;
    int us = 1;
    for (n in 2:size(t)) {
      if (t[n] != t[n - 1]) us += 1;
    }
    return us;
  }
```

This is then used to compute the value `J` to send into the function that computes the position in the array of failure times where each new failure time starts, plus an end point that goes one past the target. This is a standard way in Stan to code ragged arrays.

```
  array[] int unique_starts(vector t, int J) {
    array[J + 1] int starts;
    if (J == 0) return starts;
    starts[1] = 1;
    int pos = 2;
    for (n in 2:size(t)) {
      if (t[n] != t[n - 1]) {
    starts[pos] = n;
    pos += 1;
      }
    }
    starts[J + 1] = size(t) + 1;
    return starts;
  }
}
```

The data format is exactly the same as for the model in the previous section, but in this case, the transformed data block is used to cache some precomputations required for the model, namely the ragged array grouping elements that share the same survival time.

```
transformed data {
  int<lower=0> J = num_unique_starts(t);
  array[J + 1] int<lower=0> starts = unique_starts(t, J);
}
```

For each unique survival time `j` in `1:J`, the subjects indexed from `starts[j]` to `starts[j + 1] - 1` (inclusive) share the same survival time. The number of elements with survival time `j` is thus `(starts[j + 1] - 1) - starts[j] + 1`, or just `starts[j + 1] - starts[j]`.

The parameters and prior are also the same—just a vector `beta` of coefficients with a centered normal prior. Although it starts with the same caching of results for later, and uses the same accumulator for the denominator, the overall partial likelihood is much more involved, and depends on the user-defined functions defining the transformed data variables `J` and `starts`.

```
  vector[N] log_theta = x * beta;
  vector[N_c] log_theta_c = x_c * beta;
  real log_denom_lhs = log_sum_exp(log_theta_c);
  for (j in 1:J) {
    int start = starts[j];
    int end = starts[j + 1] - 1;
    int len = end - start + 1;
    real log_len = log(len);
    real numerator = sum(log_theta[start:end]);
    log_denom_lhs = log_sum_exp(log_denom_lhs,
                                log_sum_exp(log_theta[start:end]));
    vector[len] diff;
    for (ell in 1:len) {
      diff[ell] = log_diff_exp(log_denom_lhs,
                               log(ell - 1) - log_len
                               + log_sum_exp(log_theta[start:end]));
    }
    target += numerator - sum(diff);
  }
```

The special function `log_diff_exp` is defined as

$$\text{logDiffExp}(u, v) = \log(\exp(u) - \exp(v)).$$

Because of how `J` and `starts` are constructed, the length `len` will always be strictly positive so that the log is well defined.

Part II

Programming Techniques

18. 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*.

18.1. 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 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*: $\approx 1.8 \times 10^{308}$

[1] The notable exception is Intel's optimizing compilers under certain optimization settings.

- *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.

18.2. 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`.

18.3. 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 one 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.

18.4. Log sum of exponentials

Working on the log scale, multiplication is converted to addition,

$$\log(a \cdot b) = \log a + \log b.$$

Thus sequences of multiplication operations can remain on the log scale. But what about addition? Given $\log a$ and $\log b$, how do we get $\log(a + b)$? Working out the algebra,

$$\log(a + b) = \log(\exp(\log a) + \exp(\log b)).$$

Log-sum-exp function

The nested log of sum of exponentials is so common, it has its own name, "log-sum-exp",

$$\text{log-sum-exp}(u, v) = \log(\exp(u) + \exp(v)).$$

so that

$$\log(a + b) = \text{log-sum-exp}(\log a, \log b).$$

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,

$$\text{log-sum-exp}(u, v) = \max(u, v) + \log\big(\exp(u - \max(u, v)) + \exp(v - \max(u, v))\big).$$

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

$$\text{log-sum-exp}(u, v) = u + \log\big(1 + \exp(v - u)\big).$$

Although the inner term may itself be evaluated using the built-in function `log1p`, there is only limited gain because $\exp(v - u)$ is only near zero when u is much larger than v, meaning the final 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

$$\log(a+b) = \log a + \operatorname{log1p}\big(\exp(\log b - \log a)\big).$$

Applying log-sum-exp to a sequence

The log sum of exponentials function may be generalized to sequences in the obvious way, so that if $v = v_1, \ldots, v_N$, then

$$\begin{aligned} \text{log-sum-exp}(v) &= \log \sum_{n=1}^{N} \exp(v_n) \\ &= \max(v) + \log \sum_{n=1}^{N} \exp(v_n - \max(v)). \end{aligned}$$

The exponent cannot overflow because its argument is either zero or negative. This form makes it easy to calculate $\log(u_1 + \cdots + u_N)$ given only $\log u_n$.

Calculating means with log-sum-exp

An immediate application is to computing the mean of a vector u entirely on the log scale. That is, given $\log u$ and returning $\log \text{mean}(u)$.

$$\begin{aligned} \log\left(\frac{1}{N}\sum_{n=1}^{N} u_n\right) &= \log\frac{1}{N} + \log\sum_{n=1}^{N}\exp(\log u_n) \\ &= -\log N + \text{log-sum-exp}(\log u). \end{aligned}$$

where $\log u = (\log u_1, \ldots, \log u_N)$ is understood elementwise.

18.5. 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.
```

19. Matrices, Vectors, Arrays, and Tuples

This chapter provides pointers as to how to choose among the various container types (matrix, vector, array, and tuple) provided by Stan.

19.1. Basic motivation

Stan provides three basic scalar types, `int`, `real`, and `complex`, as well as three basic linear algebra types, `vector`, `row_vector`, and `matrix`. Stan allows arrays of any dimensionality, containing 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.

```
array[N] real a;
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 $M \times N$.

```
array[M, N] real b;          // b[m] : array[] real     (efficient)
array[M] vector[N] b;        // b[m] : vector     (efficient)
array[M] row_vector[N] b;    // 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).

19.2. Tuple types

Arrays may contain entries of any type, but the types must be the same for all entries. Matrices and vectors contain either real numbers or complex numbers, but all the contained types are the same (e.g., if a vector has a single `complex` typed entry, all the entries are `complex`).

With arrays or vectors, we can represent pairs of real numbers or pairs of complex numbers. For example, a `complex_vector[3]` holds exactly three complex numbers. With arrays and vectors, there is no way to represent a pair consisting of an integer and a real number.

Tuples provide a way to represent a sequence of values of heterogeneous types. For example, `tuple(int, real)` is the type of a pair consisting of an integer and a real number and `tuple(array[5] int, vector[6])` is the type of pairs where the first element is a five-element array of integers, the second entry is an integer, and the third is a six-element vector.

Tuple syntax

Tuples are declared using the keyword `tuple` followed by a sequence of type declarations in parentheses. Tuples are constructed using only parentheses. The following example illustrations both declaration and construction.

```
tuple(int, vector[3]) ny = (5, [3, 2.9, 1.8]');
```

The elements of a tuple are accessed by position, starting from 1. For example, we can extract the elements of the tuple above using

```
int n = ny.1;
vector[3] y = ny.2;
```

We can also assign into the elements of a tuple.

```
tuple(int, vector[3], complex) abc;
abc.1 = 5;
abc.2[1] = 3;
abc.2[2] = 2.9;
abc.2[3] = 1.4798;
abc.3 = 2 + 1.9j;
```

As the cascaded indexing example shows, the result of `abc.1` is an lvalue (i.e., something to which values may be assigned), and we can further index into it to create new lvalues (e.g., `abc.2[1]` pulls out the first element of the vector value of the second element of the tuple.)

There are two efficiency considerations for tuples. First, like the other container types, tuples are passed to functions by constant reference, which means only a pointer gets passed rather than copying the data. Second, like the array types, creating a tuple requires copying the data for all of its elements. For example, in the following code, the matrix is copied, entailing 1000 copies of scalar values.

```
int a = 5;
matrix[10, 100] b = ...;
tuple(int, matrix[10, 100]) ab = (a, b);  // COPIES b
b[1,1] = 10.3;  // does NOT change ab
```

Applications of tuples

Tuples are primarily useful for two things. First, they provide a way to encapsulate a group of heterogeneous items so that they may be passed as a group. This lets us define arrays of structures as well as structures of arrays. For example, `array[N] tuple(int, real, vector[5])` is an array of tuples, each of which has an integer, real, and vector component. Alternatively, we can represent the same information using a tuple of parallel arrays as `tuple(array[N] int, array[N] real, array[N] vector[5])`.

The second use is for function return values. Here, if a function computes two different things with different types, and the computation shares work, it's best to write one function that returns both things. For example, an eigendecomposition returns a pair consisting of a vector of eigenvalues and a matrix of eigenvectors, whereas a singular value decomposition returns three matrices of different shapes. Before introducing tuples in version 2.33, the QR decomposition of matrix $A = Q \cdot R$, where Q is orthonormal and R is upper triangular. In the past, this required two function calls.

```
matrix[M, N] A = ...;
matrix[M, M] Q = qr_Q(A);
matrix[M, N] R = qr_R(A);
```

With tuples, this can be simplified to the following,

```
tuple(matrix[M, M], matrix[M, N]) QR = qr(A);
```

with `QR.1` being `Q` and `QR.2` giving `R`.

19.3. 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.

19.4. 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.

```
array[M, N] real a;
// ...
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

```
array[I, J] matrix[M, N] b;
// ...
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

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

is much more efficient than the pure matrix version

```
matrix[M, N] b;
// ...
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

```
array[N] row_vector[K] x;    // 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.

19.5. 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.

```
array[M] row_vector[N] a;
// ...
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.

```
array[M, N] real a;
// ...
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."

19.6. 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.

19.7. 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.

20. 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. In many cases, there are functions that provide similar behavior.

Allowing multiple indexes supports inline vectorization of models. For instance, consider the data model 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 faster than the loop version; it 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);
}
```

The boost in speed compared to the original version is because the single call to the normal log density in the distribution statement will be much more memory efficient than the original version.

20.1. 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.

```
array[3] int c;
// ... define: c == (5, 9, 7)
array[4] int idxs;
// ... define: idxs == (3, 3, 1, 2)
array[4] int d;
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.

```
array[2, 3] int c;
// ... define: c = ((1, 3, 5), ((7, 11, 13))
array[4] int idxs;
// ... define: idxs = (2, 2, 1, 2)
array[4, 3] int d
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.

```
array[4] int e;
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

```
array[2, 3] int c;
// ... define: c = ((1, 3, 5), (7, 11, 13))
array[3] int idxs1;
// ... define: idxs1 = (2, 2, 1)
array[2] int idxs2;
// ... define: idxs2 = (1, 3)
array[3, 2] int d;
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.

20.2. 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.

```
array[7] int c;
// ...
array[4] int d;
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`.

Slicing functions

Stan provides `head` and `tail` functions that pull out prefixes or suffixes of vectors, row vectors, and one-dimensional arrays. In each case, the return type is the same as the argument type. For example,

```
vector[M] a = ...;
vector[N] b = head(a, N);
```

assigns `b` to be a vector equivalent to the first `N` elements of the vector `a`. The function `tail` works the same way for suffixes, with

```
array[M] a = ...;
array[N] b = tail(a, N);
```

Finally, there is a segment function, which specifies a first element and number of elements. For example,

```
array[15] a = ...;
array[3] b = segment(a, 5, 3);
```

will set `b` to be equal to `{ a[5], a[6], a[7] }`, so that it starts at element 5 of `a` and includes a total of 3 elements.

20.3. 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.

```
array[3] int a;
array[2] int c;
array[2] int idxs;
// ... 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.

```
array[5, 7] int a;
array[2, 2] int c;
// ...
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.

```
array[10, 13] int a;
array[2] int c;
// ...
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.

```
array[3] int a;
// ... 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.

```
array[3] int a;
array[3] int idxs;
// ... 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.

20.4. 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.

```
array[3] vector[5] v;
array[7] int idxs;
// ...
vector[7] u;
u = v[2, idxs];

array[7] real w;
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.

```
array[5, 7] matrix[3, 4] m;
// ...
array[2] matrix[3, 4] a;
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.

```
// ...
array[2] row_vector[3] c;
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]`

Block, row, and column extraction for matrices

Matrix slicing can also be performed using the `block` function. For example,

```
matrix[20, 20] a = ...;
matrix[3, 2] b = block(a, 5, 9, 3, 2);
```

will set `b` equal to the submatrix of `a` starting at index [5, 9] and extending 3 rows and 2 columns. Thus `block(a, 5, 9, 3, 2)` is equivalent to `b[5:7, 9:10]`.

The `sub_col` function extracts a slice of a column of a matrix as a vector. For example,

```
matrix[10, 10] a = ...;
vector b = sub_col(a, 2, 3, 5);
```

will set `b` equal to the vector `a[2:6, 3]`, taking the element starting at [2, 3], then extending for a total of 5 rows. The function `sub_row` works the same way for extracting a slice of a row as a row vector. For example, `sub_row(a, 2, 3, 5)` is equal to the row vector `a[2, 3:7]`, which also starts at position [2, 3] then extends for a total of 5 columns.

20.5. Matrices with parameters and constants

Suppose you have a 3*x*3 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 {
  array[7, 2] int<lower=1, upper=3> idxs
    = { {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.

21. 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]

21.1. 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 = abs(x - y);
    avg_scale = (abs(x) + abs(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 and error 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.

If an error is indicative of a problem from which it is not expected to be able to recover, Stan provides a `fatal_error` statement.

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 there is no way to recover and generate the remaining parameters, so rejections cause subsequent values to be reported as NaNs. Extra care should be taken in calling functions which may reject in the generated quantities block.

Type declarations for functions

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 types in the reference manual 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]

[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.

[3] A range of built-in validation routines is coming to Stan soon! Alternatively, the `reject` statement

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.

```
array[] real baz(array[,] 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.

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

can be used to check constraints on the simplex.

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.

21.2. 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);
  // ...
}
```

21.3. 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);
  // ...
}
```

21.4. Functions implementing change-of-variable adjustments

Functions whose names end in `_jacobian` can use the `jacobian +=` statement. This can be used to implement a custom change of variables for arbitrary parameters.

For example, this function recreates the built-in `<upper=x>` transform on real numbers:

```
real upper_bound_jacobian(real x, real ub) {
  jacobian += x;
  return ub - exp(x);
}
```

It can be used as a replacement for `real<lower=ub>` as follows:

```
functions {
  // upper_bound_jacobian as above
}
data {
  real ub;
}
parameters {
  real b_raw;
}
transformed parameters {
  real b = upper_bound_jacobian(b_raw, ub);
}
model {
  b ~ lognormal(0, 1);
  // ...
}
```

21.5. 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`.

21.6. 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 tilde symbol (~) in the distribution 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).

21.7. Overloading functions

As described in the reference manual function overloading is permitted in Stan, beginning in version 2.29.

This means multiple functions can be defined with the same name as long as they accept different numbers or types of arguments. User-defined functions can also overload Stan library functions.

Warning on usage

Overloading is a powerful productivity tool in programming languages, but it can also lead to confusion. In particular, it can be unclear at first glance which version of a function is being called at any particular call site, especially with type promotion allowed between scalar types. Because of this, it is a programming best practice that overloaded functions maintain the same meaning across definitions.

For example, consider a function `triple` which has the following three signatures

```
real triple(real x);
complex triple(complex x);
array[] real triple(array[] real);
```

One should expect that all overloads of this function perform the same basic task. This should lead to definitions of these functions which would satisfy the following assumptions that someone reading the program would expect

```
// The function does what it says
triple(3.0) == 9.0
// It is defined reasonably for different types
triple(to_complex(3.0)) == to_complex(triple(3.0))
// A container version of this function works by element
triple({3.0, 4.0})[0] == triple({3.0, 4.0}[0])
```

Note that none of these properties are enforced by Stan, they are mentioned merely to warn against uses of overloading which cause confusion.

Function resolution

Stan resolves overloaded functions by the number and type of arguments passed to the function. This can be subtle when multiple signatures with the same number of arguments are present.

Consider the following function signatures

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

Given these, the function call `foo(1.5, 2.5)` is unambiguous - it must resolve to the second signature. But, the function call `foo(1, 1.5)` could be valid for *either* under Stan's promotion rules, which allow integers to be promoted to real numbers.

To resolve this, Stan selects the signature which requires the fewest number of promotions for a given function call. In the above case, this means the call `foo(1, 1.5)` would select the first signature, because it requires 0 promotions (the second signature would require 1 promotion).

Furthermore, there must be only one such signature, e.g., the minimum number of promotions must be a unique minimum. This requirement forbids certain kinds of overloading. For example, consider the function signatures

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

These signatures do not have a unique minimum number of promotions for the call `bar(1, 2)`. Both signatures require one `int` to `real` promotion, and so it cannot be determined which is correct. Stan will produce a compilation error in this case.

Promotion from integers to complex numbers is considered to be two separate promotions, first from `int` to `real`, then from `real` to `complex`. This means that integer arguments will "prefer" a signature with real types over complex types.

For example, consider the function signatures

```
real pop(real x);
real pop(complex x);
```

Stan will select the first signature when `pop` is called with an integer argument such as `pop(0)`.

21.8. 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.

21.9. 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`.

[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 distribution statements) raises an exception via the reject statement.

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 distribution statements.

Only functions ending in `_lp` may access the log probability accumulator through distribution 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`.

21.10. 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) {
  if (n == 0) {
    return diag_matrix(rep_vector(1, rows(a)));
  } else {
    return a *  matrix_pow(a, n - 1);
  }
}
```

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;
}
```

21.11. 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 $u \sim \text{uniform}(0,1)$, then run through the inverse cumulative distribution function, $y = \text{logit}(u)$. 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 $\text{normal}(\mu, \sigma)$ variates, it is enough to generate a uniform variate $u \sim \text{uniform}(0,1)$, 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_inf(mu)) {
    reject("my_normal_rng: mu must be finite; ",
           "found mu = ", mu);
  }
  if (is_nan(sigma) || is_inf(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,

[5] The original code and impetus for including this in the manual came from the Stan forums post; by user `lcomm`, who also explained truncation above and below.

```
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),
                                          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.

22. 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.

22.1. Examples

Triangle distribution

A simple example is the triangle distribution, whose density is shaped like an isosceles triangle with corners at specified bounds and height determined by the constraint that a density integrate to 1. If $\alpha \in \mathbb{R}$ and $\beta \in \mathbb{R}$ are the bounds, with $\alpha < \beta$, then $y \in (\alpha, \beta)$ has a density defined as follows.

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

If $\alpha = -1$, $\beta = 1$, and $y \in (-1, 1)$, this reduces to

$$\text{triangle}(y \mid -1, 1) = 1 - |y|.$$

Consider the following Stan implementation of $\text{triangle}(-1, 1)$ for sampling.

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

The single scalar parameter `y` is declared as lying in the interval `(-1,1)`. The total log probability is incremented with the joint log probability of all parameters, i.e., $\log \text{Triangle}(y \mid -1, 1)$. This value is coded in Stan as `log1m(abs(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 - abs(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 distribution statement.

```
y ~ exponential(lambda);
```

There are two notable differences. First, the distribution 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 distribution 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 distribution 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

$$\texttt{binormal\_cdf}(z_1, z_2, \rho) = \Pr[Z_1 \leq z_1 \text{ and } Z_2 \leq z_2]$$

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 = abs(rho) < 1.0 ? sqrt((1 + rho) * (1 - rho))
                                : not_a_number();
    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, a2);
  }
  return 0.25 + asin(rho) / (2 * pi());
}
```

23. Proportionality Constants

When evaluating a likelihood or prior as part of the log density computation in MCMC, variational inference, or optimization, it is usually only necessary to compute the functions up to a proportionality constant (or similarly compute log densities up to an additive constant). In MCMC this comes from the fact that the distribution being sampled does not need to be normalized (and so it is the normalization constant that is ignored). Similarly the distribution does not need normalized to perform variational inference or do optimizations. The advantage of working with unnormalized distributions is they can make computation quite a bit cheaper.

There are three different syntaxes to build the model in Stan. The way to select between them is by determining if the proportionality constants are necessary. If performance is not a problem, it is always safe to use the normalized densities.

The distribution statement (~) and log density increment statement (`target +=`) with `_lupdf()` use unnormalized densities for x (dropping proportionality constants):

```
x ~ normal(0, 1);
target += normal_lupdf(x | 0, 1); // the 'u' is for unnormalized
```

The log density increment statement (`target +=`) with `_lpdf()` uses the full normalized density for x (dropping no constants):

```
target += normal_lpdf(x | 0, 1);
```

For discrete distributions, the `target +=` syntax is using `_lupmf` and `_lpmf` instead:

```
y ~ bernoulli(0.5);
target += bernoulli_lupmf(y | 0.5);
target += bernoulli_lpmf(y | 0.5);
```

23.1. Dropping Proportionality Constants

If a density $p(\theta)$ can be factored into $Kg(\theta)$ where K are all the factors that are a not a function of θ and $g(\theta)$ are all the terms that are a function of θ, then it is said that $g(\theta)$ is proportional to $p(\theta)$ up to a constant.

The advantage of all this is that sometimes K is expensive to compute and if it is not a function of the distribution that is to be sampled (or optimized or approximated with variational inference), there is no need to compute it because it will not affect the results.

Stan takes advantage of the proportionality constant fact with the ~ syntax. Take for instance the normal data model:

```
data {
  real mu;
  real<lower=0.0> sigma;
}
parameters {
  real x;
}
model {
  x ~ normal(mu, sigma);
}
```

Syntactically, this is just shorthand for the equivalent model that replaces the ~ syntax with a `target +=` statement and a `normal_lupdf` function call:

```
data {
  real mu;
  real<lower=0.0> sigma;
}
parameters {
  real x;
}
model {
  target += normal_lupdf(x | mu, sigma)
}
```

The function `normal_lupdf` is only guaranteed to return the log density of the normal distribution up to a proportionality constant density to be sampled. The proportionality constant itself is not defined. The full log density of the statement here is:

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

Now because the density here is only a function of x, the additive terms in the log density that are not a function of x can be dropped. In this case it is enough to know only the quadratic term:

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

23.2. Keeping Proportionality Constants

In the case that the proportionality constants were needed for a normal log density the function `normal_lpdf` can be used. For clarity, if there is ever a situation where it is unclear if the normalization is necessary, it should always be safe to include it. Only use the ~ or `target += normal_lupdf` syntaxes if it is absolutely clear that the proportionality constants are not necessary.

23.3. User-defined Distributions

When a custom `_lpdf` or `_lpmf` function is defined, the compiler will automatically make available a `_lupdf` or `_lupmf` version of the function. It is only possible to define custom distributions in the normalized form in Stan. Any attempt to define an unnormalized distribution directly will result in an error.

The difference in the normalized and unnormalized versions of custom probability functions is how probability functions are treated inside these functions. Any internal unnormalized probability function call will be replaced with its normalized equivalent if the normalized version of the parent custom distribution is called.

The following code demonstrates the different behaviors:

```
functions {
  real custom1_lpdf(x) {
    return normal_lupdf(x | 0.0, 1.0)
  }
  real custom2_lpdf(x) {
    return normal_lpdf(x | 0.0, 1.0)
  }
}
parameters {
  real mu;
}
model {
  mu ~ custom1(); // Normalization constants dropped
```

```
  target += custom1_lupdf(mu); // Normalization constants dropped
  target += custom1_lpdf(mu);  // Normalization constants kept

  mu ~ custom2();  // Normalization constants kept
  target += custom2_lupdf(mu);  // Normalization constants kept
  target += custom2_lpdf(mu);  // Normalization constants kept
}
```

23.4. Limitations on Using `_lupdf` and `_lupmf` Functions

To avoid ambiguities in how the normalization constants work, functions ending in `_lupdf` and `_lupmf` can only be used in the model block or user-defined probability functions (functions ending in `_lpdf` or `_lpmf`).

24. 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.

24.1. 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 data model

$$y_n \sim \mathsf{normal}(\lambda_1 + \lambda_2, \sigma).$$

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 $\lambda_1 + q, \lambda_2 - q$ 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 $\lambda_2 = -1\,000\,000\,000$

[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.

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 data model

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

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.

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 data model

$$y_n \sim \mathsf{normal}(x_n \beta, \sigma)$$

for an N-dimensional observation vector y, an $N \times K$ predictor matrix x, and a K-dimensional coefficient vector β.

[2] The marginal posterior $p(\sigma \mid y)$ for σ is proper here as long as there are at least two distinct data points.

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 model

$$y_{i,j} \sim \text{Bernoulli}(\text{logit}^{-1}(\delta_i(\alpha_j - \beta_i))).$$

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.

[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.

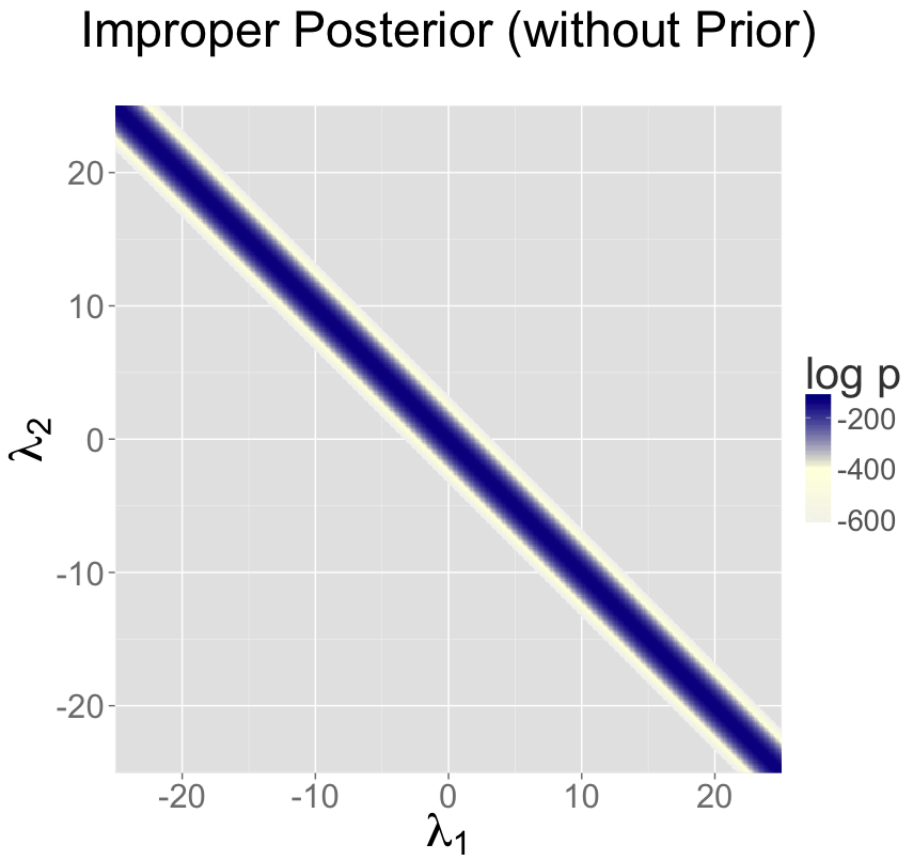


Figure 24.1: Two intercepts with improper prior

Adding a standard normal prior for the intercepts results in a proper posterior.

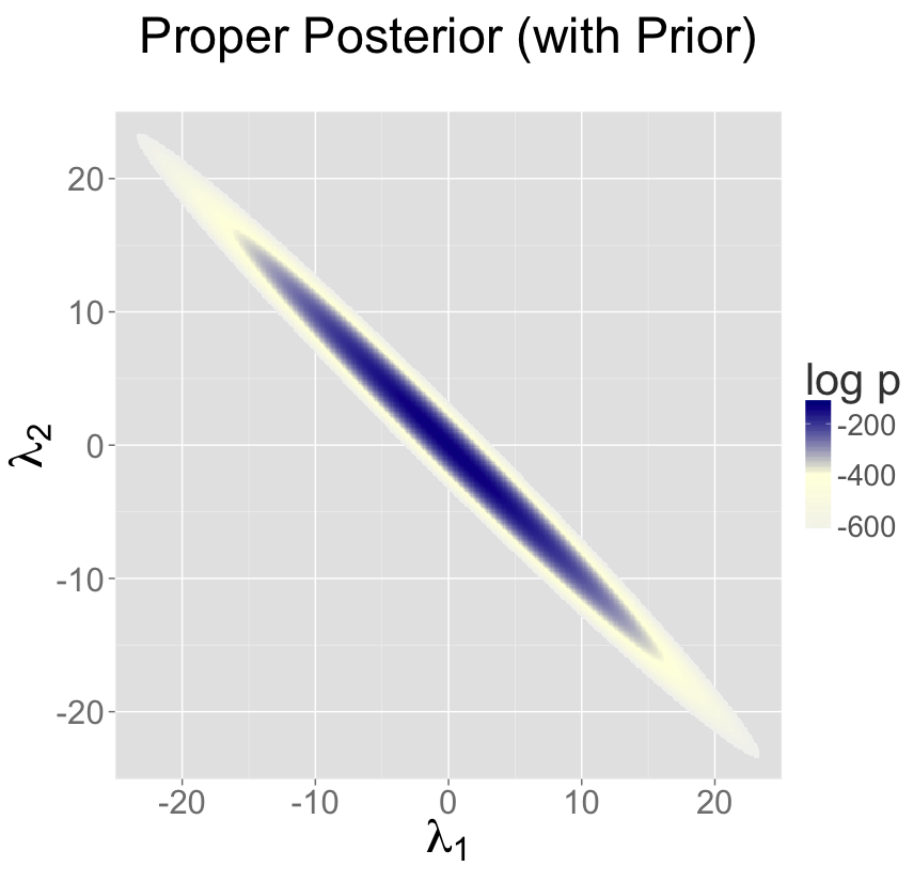


Figure 24.2: Two intercepts with proper prior

The single intercept parameterization with no prior also has a proper posterior.

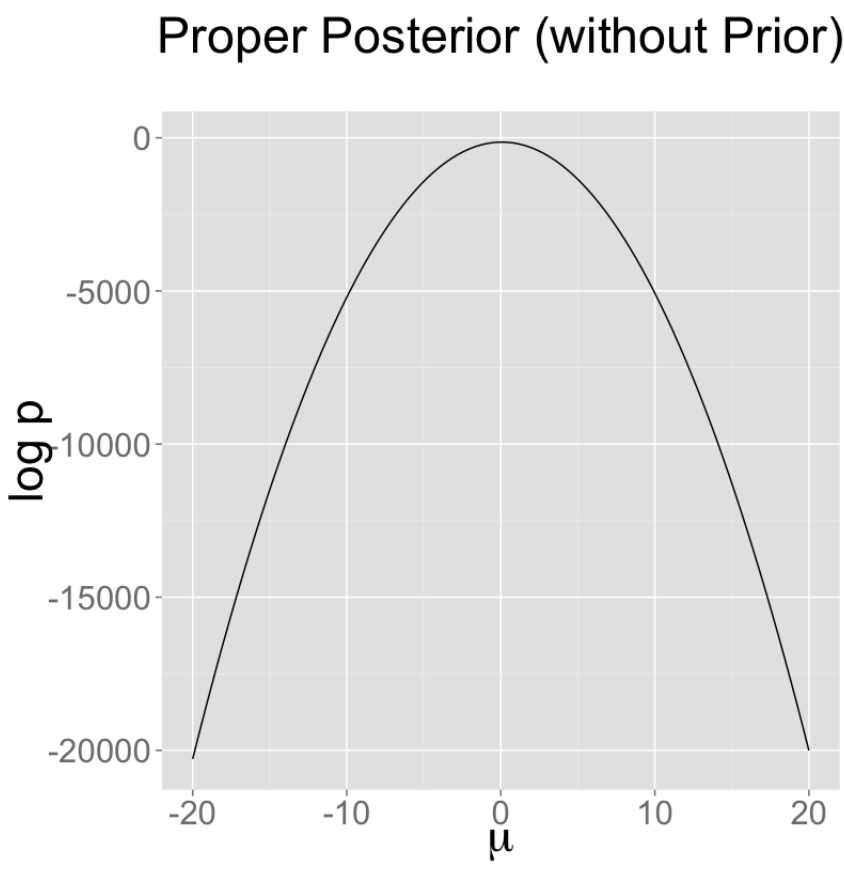


Figure 24.3: Single intercepts with improper prior

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 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.

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.

24.2. 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 data model

$$p(y \mid \theta, \mu_1, \mu_2, \sigma) = \prod_{n=1}^{N} \big(\theta \, \mathsf{normal}(y_n \mid \mu_1, \sigma) + (1 - \theta) \, \mathsf{normal}(y_n \mid \mu_2, \sigma)\big).$$

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.

24.3. 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

[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).

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).

24.4. 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 data 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.

24.5. 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 data model

$$y_n \sim \text{Bernoulli}(\text{logit}^{-1}(x_n\beta)).$$

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 $\text{logit}^{-1}(x_n\beta) \to 1$.

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.

24.6. Uniform posteriors

Suppose your model includes a parameter ψ that is defined on $[0, 1]$ and is given a flat prior $\text{uniform}(\psi \mid 0, 1)$. Now if the data don't tell us anything about ψ, the posterior is also $\text{uniform}(\psi \mid 0, 1)$.

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 $[0, 1]$.

24.7. 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 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  R_ha
lp__           -5.3e+01  7.0e-02  8.5e-01  -5.5e+01  -
5.3e+01    150       11    1.0
n_leapfrog__    1.4e+03  1.7e+01  9.2e+02   3.0e+00   2.0e+03   2987      212    1.
lambda1         1.3e+03  1.9e+03  2.7e+03  -2.3e+03   6.0e+03    2.1     0.15    5.
lambda2        -1.3e+03  1.9e+03  2.7e+03  -6.0e+03   2.3e+03    2.1     0.15    5.
sigma           1.0e+00  8.5e-03  6.2e-02   9.5e-
01   1.2e+00     54      3.9    1.1
mu              1.6e-01  1.9e-03  1.0e-01  -8.3e-03   3.3e-
01   2966      211    1.0
```

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_hat
lp__             -54  4.9e-02  1.3e+00  -5.7e+01    -
53    728      421    1.0
n_leapfrog__      157  2.8e+00  1.5e+02   3.0e+00    511   3085     1784    1.0
lambda1          0.31  2.8e-01  7.1e+00  -1.2e+01     12    638      369    1.0
lambda2         -0.14  2.8e-01  7.1e+00  -1.2e+01     12    638      369    1.0
sigma             1.0  2.6e-03  8.0e-02   9.2e-01    1.2    939      543    1.0
mu               0.16  1.8e-03  1.0e-01  -8.1e-03   0.33   3289     1902    1.0
```

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  R_hat
lp__             -54  2.5e-02    0.91  -5.5e+01   -53    -
53   1318    17198    1.0
n_leapfrog__     3.2  2.7e-01     1.7   1.0e+00   3.0   7.0     39      507    1.0
mu              0.17  2.1e-03    0.10  -3.8e-03  0.17  0.33   2408    31417    1.0
sigma            1.0  1.6e-03   0.071   9.3e-01   1.0   1.2   2094    27321    1.0
```

On the top is the non-identified model with improper uniform priors and data model $y_n \sim \text{normal}(\lambda_1 + \lambda_2, \sigma)$.

In the middle is the same data model as in top plus priors $\lambda_k \sim \text{normal}(0, 10)$.

On the bottom is an identified model with an improper prior, with data model $y_n \sim \text{normal}(\mu, \sigma)$. 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;
  array[N] real y;
}
```

```
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;
  array[N] real y;
}
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.

25. 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.

25.1. 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 $\mathbb{R}^N \to \mathbb{R}^+$. 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 Andrew 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.

25.2. 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.

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

[1] This is in contrast to (penalized) maximum likelihood estimates, which are not parameterization invariant.

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;
  array[N] simplex[K] theta;
  // ...
}
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;
  array[N] simplex[K] theta;
  // ...
}
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 $\text{uniform}(0,1)$ distribution, then $\text{logit}(u)$ is distributed as $\text{logistic}(0,1)$. 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 $\text{uniform}(0,1)$ distribution, then $\Phi^{-1}(u)$ has a $\text{normal}(0,1)$ distribution. The other way around, if v has a $\text{normal}(0,1)$ distribution, then $\Phi(v)$ has a $\text{uniform}(0,1)$ 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.

25.3. 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. Multivariate changes of variables must be injective and differentiable everywhere in their support, and they must map $\mathbb{R}^N \to \mathbb{R}^N$.

The probability must be scaled by a *Jacobian adjustment* equal to the absolute determinant of the Jacobian of the transform. In the univariate case, the Jacobian adjustment is simply 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);
  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 distribution statement, such as

[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.

```
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 defining `y_inv` to have 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 defining `y_inv` to have 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
  jacobian += -2 * log(y_inv); // Jacobian adjustment
}
model {
  y ~ gamma(2,4);
}
```

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 absolute determinant 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] ...
  jacobian += log(abs(determinant(J)));
  // ...
}
model {
  v ~ // ...
  // ...
}
```

If the determinant of 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] ...
  jacobian += sum(log(J_diag));
  // ...
}
```

25.4. Vectors with varying bounds

Stan allows scalar and non-scalar upper and lower bounds to be declared in the constraints for a container data type. 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=L>[N] alpha_raw;
  // ...
}
```

The above is equivalent to manually calculating the vector bounds by the following.

```
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 to implement the transform using an array or vector of parameters as bounds (with the requirement that the type of the variable must match the bound type) in the following.

```
data {
  int N;
  vector[N] L;  // lower bounds
  // ...
}
parameters {
  vector<lower=0>[N] alpha_raw;
  vector<lower=L + alpha_raw>[N] alpha;
  // ...
}
```

This is equivalent to 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);
  jacobian += sum(alpha_raw); // log Jacobian
```

```
  // ...
}
model {
  // ...
}
```

The adjustment in 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 $(0, 1)$. This is easily accomplished as the following.

```
data {
  int N;
  vector[N] L;  // lower bounds
  vector[N] U;  // upper bounds
  // ...
}
parameters {
  vector<lower=L, upper=U>[N] alpha;
  // ...
}
```

The same may be accomplished by manually constructing the transform as follows.

```
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 $U - L$.

26. 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.

26.1. 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).

26.2. 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.

26.3. 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.

26.4. Model conditioning and curvature

Because Stan's algorithms 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.

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 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.

26.5. 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.

26.6. 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.

26.7. 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

[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.

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.

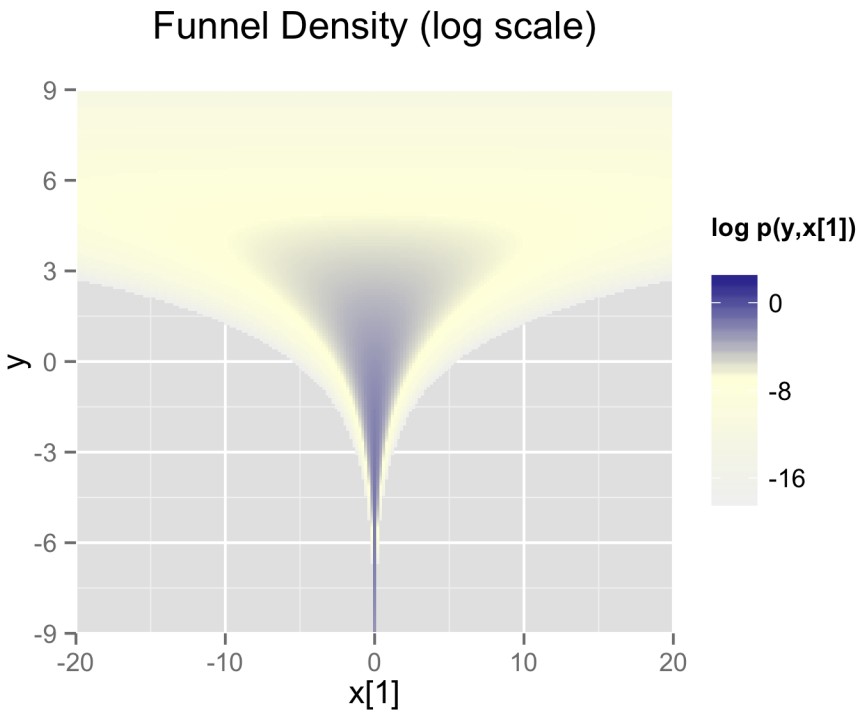


Figure 26.1: Neal's funnel density

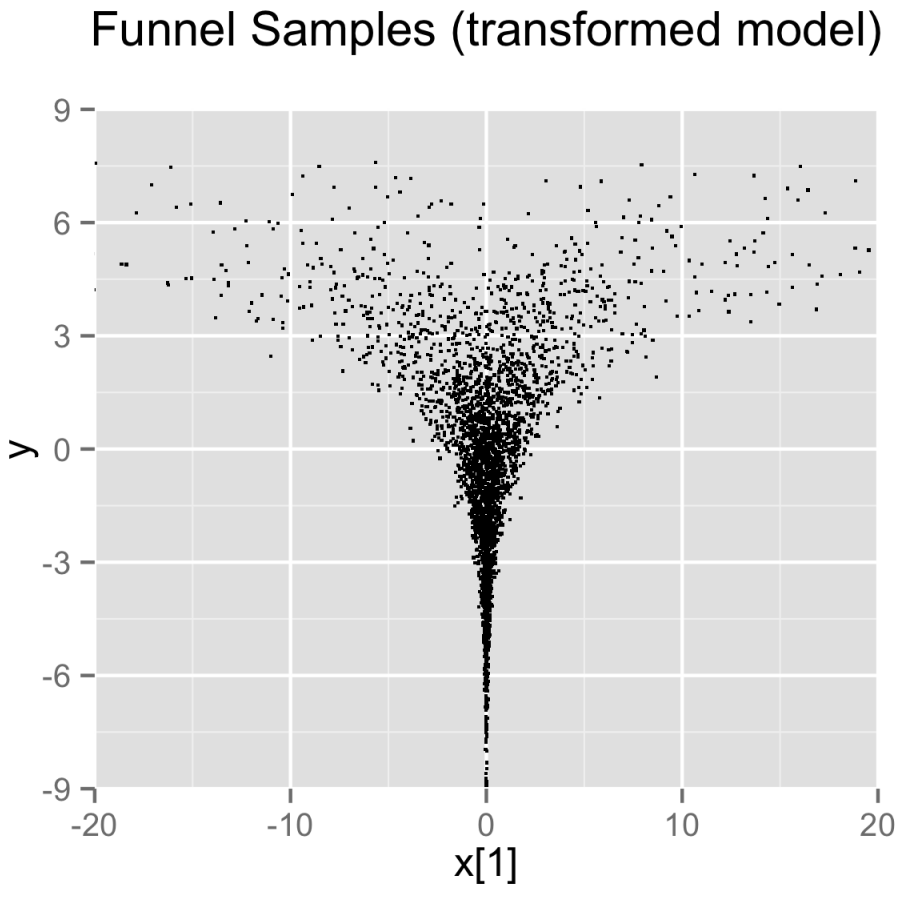
The funnel can be implemented directly in Stan as follows.

```
parameters {
  real y;
  vector[9] x;
}
model {
  y ~ normal(0, 3);
  x ~ normal(0, exp(y/2));
}
```

When the model is expressed this way, Stan has trouble sampling from the neck of the funnel, where y is small and thus x is constrained to be near 0. This is due to

the fact that the density's scale changes with y, so that a step size that works well in the body will be too large for the neck, and a step size that works in the neck will be 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 $p(y) = \text{normal}(y \mid 0, 3)$, 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.

```
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.

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 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.,

$$\mathsf{Student\text{-}t}(\beta \mid \nu, 0, 1) = \int_0^\infty \mathsf{normal}\left(\beta \middle| 0, \tau^{-0.5}\right) \times \mathsf{Gamma}\left(\tau | \nu/2, \nu/2\right) \, \mathrm{d}\tau.$$

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, the non-centered parameterization 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 {
  real mu_beta;
  real<lower=0> sigma_beta;
  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.

Alternatively, Stan's affine transform can be used to decouple `sigma` and `beta`:

```
parameters {
  real mu_beta;
  real<lower=0> sigma_beta;
  vector<offset=mu_beta, multiplier=sigma_beta>[K] beta;
  // ...
}
model {
  beta ~ normal(mu_beta, sigma_beta);
  // ...
}
```

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 {
  array[K] vector[beta] z;
  // ...
}
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)
  // ...
}
```

26.8. 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 parameters or transformed parameters.[2] 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

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

surprisingly, it will also use substantially less memory overall than incrementing `total` within the loop.

Vectorization through matrix operations

The following program directly encodes a linear regression with fixed unit noise using a two-dimensional array `x` of predictors, an array `y` of outcomes, and an array `beta` of regression coefficients.

```
data {
  int<lower=1> K;
  int<lower=1> N;
  array[K, N] real x;
  array[N] real y;
}
parameters {
  array[K] real beta;
}
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;
  array[N] vector[K] x;
  array[N] real y;
}
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.[3]

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 {
  // ...
  array[N] row_vector[K] x;
  // ...
}
parameters {
  vector[K] beta;
}
```

[3]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.

```
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 {
  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.

26.9. 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. This section provides examples for Bernoulli and normal distributions, but the same approach can be applied to other members of the exponential family.

Bernoulli sufficient statistics

Consider the following Bernoulli sampling model.

```
data {
  int<lower=0> N;
  array[N] int<lower=0, upper=1> y;
  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.

The same trick works for combining multiple binomial observations.

Normal sufficient statistics

Consider the following Stan model for fitting a normal distribution to data.

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

With the vectorized form used for `y`, Stan is clever enough to only evaluate

`log(sigma)` once, but it still has to evaluate the normal for all of `y[1]` to `y[N]`, which involves adding up all the squared differences from the mean and then dividing by `sigma` squared.

An equivalent density to the one above (up to normalizing constants that do not depend on parameters), is given in the following Stan program.

```
data {
  int N;
  vector[N] y;
}
transformed data {
  real mean_y = mean(y);
  real<lower=0> var_y = variance(y);
  real nm1_over2 = 0.5 * (N - 1);
  real sqrt_N = sqrt(N);
}
parameters {
  real mu;
  real<lower=0> sigma;
}
model {
  mean_y ~ normal(mu, sigma / sqrt_N);
  var_y ~ gamma(nm1_over2, nm1_over2 / sigma^2);
}
```

The data and parameters are the same in this program as in the first. The second version adds a transformed data block to compute the mean and variance of the data, which are the sufficient statistics here. These are stored along with two other useful constants. Then the program can define distributions over the mean and variance, both of which are scalars here.

The original Stan program and this one define the same model in the sense that they define the same log density up to a constant additive term that does not depend on the parameters. The priors on `mu` and `sigma` are both improper, but proper priors could be added as additional statements in the model block without affecting the sufficiency.

This transform explicitly relies on aggregating the data. Using this trick on parameters leads to more computation than just computing the normal log density, even before accounting for the non-linear change of variables in the variance.

Poisson sufficient statistics

The Poisson distribution is the easiest case, because the sum of observations is sufficient. Specifically, we can replace

```
y ~ poisson(lambda);
```

with

```
sum(y) ~ poisson(size(y) * lambda);
```

This will work even if `y` is a parameter vector because no Jacobian adjustment is required for summation.

26.10. 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]]`.

26.11. Exploiting conjugacy

Continuing the model from the previous section, the conjugacy of the beta prior and binomial 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)`.

26.12. Standardizing predictors

Standardizing the data so that all predictors have a zero sample mean and unit sample variance has the following potential benefits:

- It helps in faster convergence of MCMC chains.
- It makes the model less sensitive to the specifics of the parameterization.
- It aids in the interpretation and comparison of the importance of coefficients across different predictors.

When there are large differences between the units and scales of the predictors, standardizing the predictors is especially useful. This section illustrates the principle for a simple linear regression.

Suppose that $y = (y_1, \ldots, y_N)$ is a vector of N outcomes and $x = (x_1, \ldots, x_N)$ the corresponding vector 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 x has very large or very small values or if the mean of the values is far away from 0 (on the scale of the values), then it can be more efficient to standardize the predictor values x_n. First the elements of x are zero-centered by subtracting the mean, then scaled by dividing by the standard deviation.

The mean of x is given by:

$$mean_x = \frac{1}{N} \sum_{n=1}^{N} x_n$$

The standard deviation of x is calculated as:

$$sd_x = \left(\frac{1}{N} \sum_{n=1}^{N} (x_n - mean_x)^2 \right)^{1/2}$$

With these, we compute the z, the standardized predictors

$$z_n = \frac{x_n - mean_x}{sd_x}$$

where z_n is the standardized value corresponding to x_n.

The inverse transform is defined by reversing the two normalization steps, first rescaling by the same deviation and relocating by the sample mean.

$$x_n = z_n sd_x + mean_x$$

Standardizing the predictors standardizes the scale of the variables, and hence 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 ~ normal(0, 5);
  // likelihood
  y ~ normal(x * beta + alpha, sigma);
}
```

The data block for the standardized model is identical. The mean and standard deviation of the data are defined in the transformed data block, along with the standardized predictors.

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

```
  real mean_x = mean(x);
  real sd_x = sd(x);
  vector[N] x_std = (x - mean_x) / sd_x;
}
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 ~ normal(0, 5);
  y ~ normal(x_std * beta_std + alpha_std, sigma_std);
}
```

The parameters are renamed to indicate that they aren't the "natural" parameters. The transformed data `x_std` is defined in terms of variables `mean_x` and `sd_x`; by declaring these variables in the `transformed data` block, they will be available in all following blocks, and therefore can be used in the `generated quantities` block to record the "natural" parameters `alpha` and `beta`.

The fairly diffuse priors on the coefficients 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 original regression

$$y_n = \alpha + \beta x_n + \epsilon_n$$

has been transformed to a regression on the standardized data variable z,

$$y_n = \alpha' + \beta' z_n + \epsilon_n.$$

The likelihood is specified in terms of the standardized parameters. The original slope β is the standardized slope β' scaled by the inverse of the standard deviation of x. The original intercept α is the intercept from the standardized model α', corrected for the effect of scaling and centering x. Thus, the formulas to retrieve α and β from α' and β' are:

$$\beta = \frac{\beta'}{\sigma_x}$$
$$\alpha = \alpha' - \beta' \frac{\mu_x}{\sigma_x}$$

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 beta = beta_std / sd_x;
  real alpha = alpha_std - beta_std * mean_x / sd_x;

}
```

When there are multiple real-valued predictors, i.e., when `K` is the number of predictors, `x` is an $N \times K$ matrix, and `beta` ia K-vector of coefficients, then `x * beta` is an N-vector of predictions, one for each of the N data items. When $K \ll N$ the QR reparameterization is recommended for linear and generalized linear models unless there is an informative prior on the location of β.

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.

26.13. 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.

27. Parallelization

Stan has support for different types of parallelization: multi-threading with Intel Threading Building Blocks (TBB), multi-processing with Message Passing Interface (MPI) and manycore processing with OpenCL.

Multi-threading in Stan can be used with two mechanisms: reduce with summation and rectangular map. The latter can also be used with multi-processing.

The advantages of reduce with summation are:

1. More flexible argument interface, avoiding the packing and unpacking that is necessary with rectanguar map.
2. Partitions data for parallelization automatically (this is done manually in rectanguar map).
3. Is easier to use.

The advantages of rectangular map are:

1. Returns a list of vectors, while the reduce summation returns only a scalar.
2. Can be parallelized across multiple cores and multiple computers, while reduce summation can only parallelized across multiple cores on a single machine.

The actual speedup gained from using these functions will depend on many details. It is strongly recommended to only parallelize the computationally most expensive operations in a Stan program. Oftentimes this is the evaluation of the log likelihood for the observed data. When it is not clear which parts of the model is the most computationally expensive, we recommend using profiling, which is available in Stan 2.26 and newer.

Since only portions of a Stan program will run in parallel, the maximal speedup one can achieve is capped, a phenomen described by Amdahl's law.

27.1. Reduce-sum

It is often necessary in probabilistic modeling to compute the sum of a number of independent function evaluations. This occurs, for instance, when evaluating a number of conditionally independent terms in a log-likelihood. If `g: U -> real` is the function and `{ x1, x2, ... }` is an array of inputs, then that sum looks like:

```
g(x1) + g(x2) + ...
```

`reduce_sum` and `reduce_sum_static` are tools for parallelizing these calculations.

For efficiency reasons the reduce function doesn't work with the element-wise evaluated function g, but instead the partial sum function `f: U[] -> real`, where `f` computes the partial sum corresponding to a slice of the sequence `x` passed in. Due to the associativity of the sum reduction it holds that:

```
g(x1) + g(x2) + g(x3) = f({ x1, x2, x3 })
                      = f({ x1, x2 }) + f({ x3 })
                      = f({ x1 }) + f({ x2, x3 })
                      = f({ x1 }) + f({ x2 }) + f({ x3 })
```

With the partial sum function `f: U[] -> real` reduction of a large number of terms can be evaluated in parallel automatically, since the overall sum can be partitioned into arbitrary smaller partial sums. The exact partitioning into the partial sums is not under the control of the user. However, since the exact numerical result will depend on the order of summation, Stan provides two versions of the reduce summation facility:

- `reduce_sum`: Automatically choose partial sums partitioning based on a dynamic scheduling algorithm.
- `reduce_sum_static`: Compute the same sum as `reduce_sum`, but partition the input in the same way for given data set (in `reduce_sum` this partitioning might change depending on computer load).

`grainsize` is the one tuning parameter. For `reduce_sum`, `grainsize` is a suggested partial sum size. A `grainsize` of 1 leaves the partitioning entirely up to the scheduler. This should be the default way of using `reduce_sum` unless time is spent carefully picking `grainsize`. For picking a `grainsize`, see details below.

For `reduce_sum_static`, `grainsize` specifies the maximal partial sum size. With `reduce_sum_static` it is more important to choose `grainsize` carefully since it entirely determines the partitioning of work. See details below.

For efficiency and convenience additional shared arguments can be passed to every term in the sum. So for the array `{ x1, x2, ... }` and the shared arguments `s1, s2, ...`stan the effective sum (with individual terms) looks like:

```
g(x1, s1, s2, ...) + g(x2, s1, s2, ...) + g(x3, s1, s2, ...) + ...
```

which can be written equivalently with partial sums to look like:

```
f({ x1, x2 }, s1, s2, ...) + f({ x3 }, s1, s2, ...)
```

where the particular slicing of the x array can change.

Given this, the signatures are:

```
real reduce_sum(F f, array[] T x, int grainsize, T1 s1, T2 s2, ...)
real reduce_sum_static(F f, array[] T x, int grainsize, T1 s1, T2 s2, ...)
```

1. f - User defined function that computes partial sums
2. x - Array to slice, each element corresponds to a term in the summation
3. grainsize - Target for size of slices
4. s1, s2, ... - Arguments shared in every term

The user-defined partial sum functions have the signature:

```
real f(array[] T x_slice, int start, int end, T1 s1, T2 s2, ...)
```

and take the arguments:

1. x_slice - The subset of x (from reduce_sum / reduce_sum_static) for which this partial sum is responsible (x_slice = x[start:end])
2. start - An integer specifying the first term in the partial sum
3. end - An integer specifying the last term in the partial sum (inclusive)
4. s1, s2, ... - Arguments shared in every term (passed on without modification from the reduce_sum / reduce_sum_static call)

The user-provided function f is expected to compute the partial sum with the terms start through end of the overall sum. The user function is passed the subset x[start:end] as x_slice. start and end are passed so that fstan can index any of the tailing sM arguments as necessary. The trailing sM arguments are passed without modification to every call of f.

A reduce_sum (or reduce_sum_static) call:

```
real sum = reduce_sum(f, x, grainsize, s1, s2, ...);
```

can be replaced by either:

```
real sum = f(x, 1, size(x), s1, s2, ...);
```

or the code:

```
real sum = 0.0;
for(i in 1:size(x)) {
```

```
  sum += f({ x[i] }, i, i, s1, s2, ...);
}
```

Example: logistic regression

Logistic regression is a useful example to clarify both the syntax and semantics of reduce summation and how it can be used to speed up a typical model. A basic logistic regression can be coded in Stan as:

```
data {
  int N;
  array[N] int y;
  vector[N] x;
}
parameters {
  vector[2] beta;
}
model {
  beta ~ std_normal();
  y ~ bernoulli_logit(beta[1] + beta[2] * x);
}
```

In this model predictions are made about the `N` outputs `y` using the covariate `x`. The intercept and slope of the linear equation are to be estimated. The key point to getting this calculation to use reduce summation, is recognizing that the statement:

```
y ~ bernoulli_logit(beta[1] + beta[2] * x);
```

can be rewritten (up to a proportionality constant) as:

```
for(n in 1:N) {
  target += bernoulli_logit_lpmf(y[n] | beta[1] + beta[2] * x[n])
}
```

Now it is clear that the calculation is the sum of a number of conditionally independent Bernoulli log probability statements, which is the condition where reduce summation is useful. To use the reduce summation, a function must be written that can be used to compute arbitrary partial sums of the total sum. Using the interface defined in Reduce-Sum, such a function can be written like:

```
functions {
  real partial_sum(array[] int y_slice,
                   int start, int end,
```

```
                    vector x,
                    vector beta) {
    return bernoulli_logit_lpmf(y_slice | beta[1] + beta[2] * x[start:end]
  }
}
```

The likelihood statement in the model can now be written:

```
target += partial_sum(y, 1, N, x, beta); // Sum terms 1 to N of the likeli
```

In this example, `y` was chosen to be sliced over because there is one term in the summation per value of `y`. Technically `x` would have worked as well. Use whatever conceptually makes the most sense for a given model, e.g. slice over independent terms like conditionally independent observations or groups of observations as in hierarchical models. Because `x` is a shared argument, it is subset accordingly with `start:end`. With this function, reduce summation can be used to automatically parallelize the likelihood:

```
int grainsize = 1;
target += reduce_sum(partial_sum, y,
                     grainsize,
                     x, beta);
```

The reduce summation facility automatically breaks the sum into pieces and computes them in parallel. `grainsize = 1` specifies that the `grainsize` should be estimated automatically. The final model is:

```
functions {
  real partial_sum(array[] int y_slice,
                   int start, int end,
                   vector x,
                   vector beta) {
    return bernoulli_logit_lpmf(y_slice | beta[1] + beta[2] * x[start:end]
  }
}
data {
  int N;
  array[N] int y;
  vector[N] x;
}
parameters {
```

```
  vector[2] beta;
}
model {
  int grainsize = 1;
  beta ~ std_normal();
  target += reduce_sum(partial_sum, y,
                       grainsize,
                       x, beta);
}
```

Picking the grainsize

The rational for choosing a sensible `grainsize` is based on balancing the overhead implied by creating many small tasks versus creating fewer large tasks which limits the potential parallelism.

In `reduce_sum`, `grainsize` is a recommendation on how to partition the work in the partial sum into smaller pieces. A `grainsize` of 1 leaves this entirely up to the internal scheduler and should be chosen if no benchmarking of other grainsizes is done. Ideally this will be efficient, but there are no guarantees.

In `reduce_sum_static`, `grainsize` is an upper limit on the worksize. Work will be split until all partial sums are just smaller than `grainsize` (and the split will happen the same way every time for the same inputs). For the static version it is more important to select a sensible `grainsize`.

In order to figure out an optimal `grainsize`, if there are `N` terms and `M` cores, run a quick test model with `grainsize` set roughly to `N / M`. Record the time, cut the `grainsize` in half, and run the test again. Repeat this iteratively until the model runtime begins to increase. This is a suitable `grainsize` for the model, because this ensures the calculations can be carried out with the most parallelism without losing too much efficiency.

For instance, in a model with `N=10000` and `M = 4`, start with `grainsize = 2500`, and sequentially try `grainsize = 1250`, `grainsize = 625`, etc.

It is important to repeat this process until performance gets worse. It is possible after many halvings nothing happens, but there might still be a smaller `grainsize` that performs better. Even if a sum has many tens of thousands of terms, depending on the internal calculations, a `grainsize` of thirty or forty or smaller might be the best, and it is difficult to predict this behavior. Without doing these halvings until performance actually gets worse, it is easy to miss this.

27.2. Map-rect

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).

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*.

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, array[] real, array[] int):vector f,
                vector phi, array[] vector thetas,
                data array[,] real x_rs, data array[,] 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, array[] real, array[] int):vector` indicates that the function argument `f` must have the following signature.

```
vector f(vector phi, vector theta,
         data array[] real x_r, data array[] 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.

Example: 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 accommodate direct translation to a mapped implementation.

```
data {
  array[12] int y;
  array[12] real x;
}
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, array[] real x, array[] int y) {
    real lp = bernoulli_logit_lpmf(y | beta[1]
                                       + to_vector(x) * beta[2]);
    return [lp]';
  }
}
data {
  array[12] int y;
  array[12] real x;
}
transformed data {
  // K = 3 shards
  array[3, 4] = { y[1:4], y[5:8], y[9:12] int ys };
  array[3, 4] = { x[1:4], x[5:8], x[9:12] real xs };
  array[3] vector[0] theta;
}
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.

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 data model for the vote $y_n \in \{0, 1\}$ is

$$y_n \sim \text{Bernoulli}\Big(\text{logit}^{-1} \left(\alpha_{s[n]} + \beta_{s[n]} \, x_n \right) \Big).$$

[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.

[2] This example is a simplified form of the model described in (Andrew Gelman and Hill 2007, sec. 14.2)

The slopes and intercepts get hierarchical priors,

$$\alpha_k \sim \text{normal}(\mu_\alpha, \sigma_\alpha)$$
$$\beta_k \sim \text{normal}(\mu_\beta, \sigma_\beta)$$

Unmapped implementation

This model can be coded up in Stan directly as follows.

```
data {
  int<lower=0> K;
  int<lower=0> N;
  array[N] int<lower=1, upper=K> kk;
  vector[N] x;
  array[N] int<lower=0, upper=1> y;
}
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]);
  }
  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 model. 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 distribution statement is also vectorized using multi-indexing with index `kk` for the states and elementwise multiplication (`.*`) for the income `x`. The vectorized distribution statement 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. In this function we can't use distribution statements, but need to accumulate the desired log prior and log likelihood terms to the return value. 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 `bernoulli_logit_glm` derives from the fact that it's a generalized linear model with a Bernoulli data model and logistic link function).

```
functions {
 vector bl_glm(vector mu_sigma, vector beta,
               array[] real x, array[] 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;
```

```
  array[N] int<lower=1, upper=K> kk;
  vector[N] x;
  array[N] int<lower=0, upper=1> y;
}
transformed data {
  int<lower=0> J = N / K;
  array[K, J] real x_r;
  array[K, J] int<lower=0, upper=1> x_i;
  {
    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 {
  array[K] vector[2] beta;
  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));
```

[3]This makes the strong assumption that each group has the same number of observations!

```
}
```

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 computation. 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.

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.

27.3. OpenCL

OpenCL (Open Computing Language) is a framework that enables writing programs that execute across heterogeneous platforms. An OpenCL program can be run on CPUs and GPUs. In order to run OpenCL programs, an OpenCL runtime be installed on the target system.

Stan's OpenCL backend is currently supported in CmdStan and its wrappers. In order to use it, the model must be compiled with the `STAN_OPENCL` makefile flag. Setting this flag means that the Stan-to-C++ translator (`stanc3`) will be supplied the `--use-opencl` flag and that the OpenCL enabled backend (Stan Math functions)

will be enabled.

In Stan, the following distributions can be automatically run in parallel on both CPUs and GPUs with OpenCL:

- bernoulli_lpmf
- bernoulli_logit_lpmf
- bernoulli_logit_glm_lpmf*
- beta_lpdf
- beta_proportion_lpdf
- binomial_lpmf
- categorical_logit_glm_lpmf*
- cauchy_lpdf
- chi_square_lpdf
- double_exponential_lpdf
- exp_mod_normal_lpdf
- exponential_lpdf
- frechet_lpdf
- gamma_lpdf
- gumbel_lpdf
- inv_chi_square_lpdf
- inv_gamma_lpdf
- logistic_lpdf
- lognormal_lpdf
- neg_binomial_lpmf
- neg_binomial_2_lpmf
- neg_binomial_2_log_lpmf
- neg_binomial_2_log_glm_lpmf*
- normal_lpdf
- normal_id_glm_lpdf*
- ordered_logistic_glm_lpmf*
- pareto_lpdf
- pareto_type_2_lpdf
- poisson_lpmf
- poisson_log_lpmf
- poisson_log_glm_lpmf*
- rayleigh_lpdf
- scaled_inv_chi_square_lpdf
- skew_normal_lpdf
- std_normal_lpdf

- student_t_lpdf
- uniform_lpdf
- weibull_lpdf

* OpenCL is not used when the covariate argument to the GLM functions is a `row_vector`.

Part III

Posterior Inference & Model Checking

28. Posterior Predictive Sampling

The goal of inference is often posterior prediction, that is evaluating or sampling from the posterior predictive distribution $p(\tilde{y} \mid y)$, where y is observed data and $\tilde{y}$ is yet to be observed data. Often there are unmodeled predictors x and $\tilde{x}$ for the observed data y and unobserved data $\tilde{y}$. With predictors, the posterior predictive density is $p(\tilde{y} \mid \tilde{x}, x, y)$. All of these variables may represent multivariate quantities.

This chapter explains how to sample from the posterior predictive distribution in Stan, including applications to posterior predictive simulation and calculating event probabilities. These techniques can be coded in Stan using random number generation in the generated quantities block. Further, a technique for fitting and performing inference in two stages is presented in a section on stand-alone generated quantities in Stan

28.1. Posterior predictive distribution

Given a full Bayesian model $p(y, \theta)$, the posterior predictive density for new data $\tilde{y}$ given observed data y is

$$p(\tilde{y} \mid y) = \int p(\tilde{y} \mid \theta) \cdot p(\theta \mid y) \, \mathrm{d}\theta.$$

The product under the integral reduces to the joint posterior density $p(\tilde{y}, \theta \mid y)$, so that the integral is simply marginalizing out the parameters θ, leaving the predictive density $p(\tilde{y} \mid y)$ of future observations given past observations.

28.2. Computing the posterior predictive distribution

The posterior predictive density (or mass) of a prediction $\tilde{y}$ given observed data y can be computed using M Monte Carlo draws

$$\theta^{(m)} \sim p(\theta \mid y)$$

from the posterior as

$$p(\tilde{y} \mid y) \approx \frac{1}{M} \sum_{m=1}^{M} p(\tilde{y} \mid \theta^{(m)}).$$

Computing directly using this formula will lead to underflow in many situations, but the log posterior predictive density, $\log p(\tilde{y} \mid y)$ may be computed using the

stable log sum of exponents function as

$$\begin{aligned}\log p(\tilde{y} \mid y) &\approx \log \frac{1}{M} \sum_{m=1}^{M} p(\tilde{y} \mid \theta^{(m)}). \\ &= -\log M + \text{log-sum-exp}_{m=1}^{M} \log p(\tilde{y} \mid \theta^{(m)}),\end{aligned}$$

where

$$\text{log-sum-exp}_{m=1}^{M} v_m = \log \sum_{m=1}^{M} \exp v_m$$

is used to maintain arithmetic precision. See the section on log sum of exponentials for more details.

28.3. Sampling from the posterior predictive distribution

Given draws from the posterior $\theta^{(m)} \sim p(\theta \mid y)$, draws from the posterior predictive $\tilde{y}^{(m)} \sim p(\tilde{y} \mid y)$ can be generated by randomly generating from the sampling distribution with the parameter draw plugged in,

$$\tilde{y}^{(m)} \sim p(y \mid \theta^{(m)}).$$

Randomly drawing $\tilde{y}$ from the data model is critical because there are two forms of uncertainty in posterior predictive quantities, aleatoric uncertainty and epistemic uncertainty. Epistemic uncertainty arises because θ is unknown and estimated based only on a finite sample of data y. Aleatoric uncertainty arises because even a known value of θ leads to uncertainty about new $\tilde{y}$ as described by the data model $p(\tilde{y} \mid \theta)$. Both forms of uncertainty show up in the factored form of the posterior predictive distribution,

$$p(\tilde{y} \mid y) = \int \underbrace{p(\tilde{y} \mid \theta)}_{\substack{\text{aleatoric} \\ \text{uncertainty}}} \cdot \underbrace{p(\theta \mid y)}_{\substack{\text{epistemic} \\ \text{uncertainty}}} \, \mathrm{d}\theta.$$

28.4. Posterior predictive simulation in Stan

Posterior predictive quantities can be coded in Stan using the generated quantities block.

Simple Poisson model

For example, consider a simple Poisson model for count data with a rate parameter $\lambda > 0$ having a gamma-distributed prior,

$$\lambda \sim \text{gamma}(1, 1).$$

The N observations $y_1, \ldots, y_N$ are modeled as Poisson distributed,

$$y_n \sim \text{poisson}(\lambda).$$

Stan code

The following Stan program defines a variable for $\tilde{y}$ by random number generation in the generated quantities block.

```
data {
  int<lower=0> N;
  array[N] int<lower=0> y;
}
parameters {
  real<lower=0> lambda;
}
model {
  lambda ~ gamma(1, 1);
  y ~ poisson(lambda);
}
generated quantities {
  int<lower=0> y_tilde = poisson_rng(lambda);
}
```

The random draw from the data model for $\tilde{y}$ is coded using Stan's Poisson random number generator in the generated quantities block. This accounts for the aleatoric component of the uncertainty; Stan's posterior sampler will account for the epistemic uncertainty, generating a new $\tilde{y}^{(m)} \sim p(y \mid \lambda^{(m)})$ for each posterior draw $\lambda^{(m)} \sim p(\theta \mid y)$.

The posterior draws $\tilde{y}^{(m)}$ may be used to estimate the expected value of $\tilde{y}$ or any of its quantiles or posterior intervals, as well as event probabilities involving $\tilde{y}$. In general, $\mathbb{E}[f(\tilde{y}, \theta) \mid y]$ may be evaluated as

$$\mathbb{E}[f(\tilde{y}, \theta) \mid y] \approx \frac{1}{M} \sum_{m=1}^{M} f(\tilde{y}^{(m)}, \theta^{(m)}),$$

which is just the posterior mean of $f(\tilde{y}, \theta)$. This quantity is computed by Stan if the value of $f(\tilde{y}, \theta)$ is assigned to a variable in the generated quantities block. That is, if we have

```
generated quantities {
  real f_val = f(y_tilde, theta);
  // ...
}
```

where the value of $f(\tilde{y}, \theta)$ is assigned to variable `f_val`, then the posterior mean of `f_val` will be the expectation $\mathbb{E}[f(\tilde{y}, \theta) \mid y]$.

Analytic posterior and posterior predictive

The gamma distribution is the conjugate prior distribution for the Poisson distribution, so the posterior density $p(\lambda \mid y)$ will also follow a gamma distribution.

Because the posterior follows a gamma distribution and the sampling distribution is Poisson, the posterior predictive $p(\tilde{y} \mid y)$ will follow a negative binomial distribution, because the negative binomial is defined as a compound gamma-Poisson. That is, $y \sim \text{negative-binomial}(\alpha, \beta)$ if $\lambda \sim \text{gamma}(\alpha, \beta)$ and $y \sim \text{poisson}(\lambda)$. Rather than marginalizing out the rate parameter λ analytically as can be done to define the negative binomial probability mass function, the rate $\lambda^{(m)} \sim p(\lambda \mid y)$ is sampled from the posterior and then used to generate a draw of $\tilde{y}^{(m)} \sim p(y \mid \lambda^{(m)})$.

28.5. Posterior prediction for regressions

Posterior predictive distributions for regressions

Consider a regression with a single predictor x_n for the training outcome y_n and $\tilde{x}_n$ for the test outcome $\tilde{y}_n$. Without considering the parametric form of any of the distributions, the posterior predictive distribution for a general regression in

$$p(\tilde{y} \mid \tilde{x}, y, x) = \int p(\tilde{y} \mid x, \theta) \cdot p(\theta \mid y, x) \, \mathrm{d}\theta \tag{28.1}$$

$$\approx \frac{1}{M} \sum_{m=1}^{M} p(\tilde{y} \mid \tilde{x}, \theta^{(m)}), \tag{28.2}$$

where $\theta^{(m)} \sim p(\theta \mid x, y)$.

Stan program

The following program defines a Poisson regression with a single predictor. These predictors are all coded as data, as are their sizes. Only the observed y values are

coded as data. The predictive quantities $\tilde{y}$ appear in the generated quantities block, where they are generated by random number generation.

```
data {
  int<lower=0> N;
  vector[N] x;
  array[N] int<lower=0> y;
  int<lower=0> N_tilde;
  vector[N_tilde] x_tilde;
}
parameters {
  real alpha;
  real beta;
}
model {
  y ~ poisson_log(alpha + beta * x);
  { alpha, beta } ~ normal(0, 1);
}
generated quantities {
  array[N_tilde] int<lower=0> y_tilde
    = poisson_log_rng(alpha + beta * x_tilde);
}
```

The Poisson distributions in both the model and generated quantities block are coded using the log rate as a parameter (that's `poisson_log` vs. `poisson`, with the suffixes defining the scale of the parameter). The regression coefficients, an intercept `alpha` and slope `beta`, are given standard normal priors.

In the model block, the log rate for the Poisson is a linear function of the training data x, whereas in the generated quantities block it is a function of the test data $\tilde{x}$. Because the generated quantities block does not affect the posterior draws, the model fits α and β using only the training data, reserving $\tilde{x}$ to generate $\tilde{y}$.

The result from running Stan is a predictive sample $\tilde{y}^{(1)}, \ldots \tilde{y}^{(M)}$ where each $\tilde{y}^{(m)} \sim p(\tilde{y} \mid \tilde{x}, x, y)$.

The mean of the posterior predictive distribution is the expected value

$$\mathbb{E}[\tilde{y} \mid \tilde{x}, x, y] = \int \tilde{y} \cdot p(\tilde{y} \mid \tilde{x}, \theta) \cdot p(\theta \mid x, y) \, \mathrm{d}\theta \tag{28.3}$$

$$\approx \frac{1}{M} \sum_{m=1}^{M} \tilde{y}^{(m)}, \tag{28.4}$$

where the $\tilde{y}^{(m)} \sim p(\tilde{y} \mid \tilde{x}, x, y)$ are drawn from the posterior predictive distribution. Thus the posterior mean of `y_tilde[n]` after running Stan is the expected value of $\tilde{y}_n$ conditioned on the training data x, y and predictor $\tilde{x}_n$. This is the Bayesian estimate for $\tilde{y}$ with minimum expected squared error. The posterior draws can also be used to estimate quantiles for the median and any posterior intervals of interest for $\tilde{y}$, as well as covariance of the $\tilde{y}_n$. The posterior draws $\tilde{y}^{(m)}$ may also be used to estimate predictive event probabilities, such as $\Pr[\tilde{y}_1 > 0]$ or $\Pr[\prod_{n=1}^{\tilde{N}}(\tilde{y}_n) > 1]$, as expectations of indicator functions.

All of this can be carried out by running Stan only a single time to draw a single sample of M draws,

$$\tilde{y}^{(1)}, \ldots, \tilde{y}^{(M)} \sim p(\tilde{y} \mid \tilde{x}, x, y).$$

It's only when moving to cross-validation where multiple runs are required.

28.6. Estimating event probabilities

Event probabilities involving either parameters or predictions or both may be coded in the generated quantities block. For example, to evaluate $\Pr[\lambda > 5 \mid y]$ in the simple Poisson example with only a rate parameter λ, it suffices to define a generated quantity

```
generated quantities {
  int<lower=0, upper=1> lambda_gt_5 = lambda > 5;
  // ...
}
```

The value of the expression `lambda > 5` is 1 if the condition is true and 0 otherwise. The posterior mean of this parameter is the event probability

$$\begin{aligned} \Pr[\lambda > 5 \mid y] &= \int \mathrm{I}(\lambda > 5) \cdot p(\lambda \mid y) \, \mathrm{d}\lambda \\ &\approx \frac{1}{M} \sum_{m=1}^{M} \mathrm{I}[\lambda^{(m)} > 5], \end{aligned}$$

where each $\lambda^{(m)} \sim p(\lambda \mid y)$ is distributed according to the posterior. In Stan, this is recovered as the posterior mean of the parameter `lambda_gt_5`.

In general, event probabilities may be expressed as expectations of indicator functions. For example,

$$\begin{aligned} \Pr[\lambda > 5 \mid y] &= \mathbb{E}[\mathrm{I}[\lambda > 5] \mid y] \\ &= \int \mathrm{I}(\lambda > 5) \cdot p(\lambda \mid y) \, \mathrm{d}\lambda \\ &\approx \frac{1}{M} \sum_{m=1}^{M} \mathrm{I}(\lambda^{(m)} > 5). \end{aligned}$$

The last line above is the posterior mean of the indicator function as coded in Stan.

Event probabilities involving posterior predictive quantities $\tilde{y}$ work exactly the same way as those for parameters. For example, if $\tilde{y}_n$ is the prediction for the n-th unobserved outcome (such as the score of a team in a game or a level of expression of a protein in a cell), then

$$\begin{aligned} \Pr[\tilde{y}_3 > \tilde{y}_7 \mid \tilde{x}, x, y] &= \mathbb{E}[I[\tilde{y}_3 > \tilde{y}_7] \mid \tilde{x}, x, y] \\ &= \int \mathrm{I}(\tilde{y}_3 > \tilde{y}_7) \cdot p(\tilde{y} \mid \tilde{x}, x, y) \, \mathrm{d}\tilde{y} \\ &\approx \frac{1}{M} \sum_{m=1}^{M} \mathrm{I}(\tilde{y}_3^{(m)} > \tilde{y}_7^{(m)}), \end{aligned}$$

where $\tilde{y}^{(m)} \sim p(\tilde{y} \mid \tilde{x}, x, y)$.

28.7. Stand-alone generated quantities and ongoing prediction

Stan's sampling algorithms take a Stan program representing a posterior $p(\theta \mid y, x)$ along with actual data x and y to produce a set of draws $\theta^{(1)}, \ldots, \theta^{(M)}$ from the posterior. Posterior predictive draws $\tilde{y}^{(m)} \sim p(\tilde{y} \mid \tilde{x}, x, y)$ can be generated by drawing

$$\tilde{y}^{(m)} \sim p(y \mid \tilde{x}, \theta^{(m)})$$

from the data model. Note that drawing $\tilde{y}^{(m)}$ only depends on the new predictors $\tilde{x}$ and the posterior draws $\theta^{(m)}$. Most importantly, neither the original data or the model density is required.

By saving the posterior draws, predictions for new data items $\tilde{x}$ may be generated whenever needed. In Stan's interfaces, this is done by writing a second Stan program

that inputs the original program's parameters and the new predictors. For example, for the linear regression case, the program to take posterior draws declares the data and parameters, and defines the model.

```
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);
  alpha ~ normal(0, 5);
  beta ~ normal(0, 1);
  sigma ~ lognormal(0, 0.5);
}
```

A second program can be used to generate new observations. This follow-on program need only declare the parameters as they were originally defined. This may require defining constants in the data block such as sizes and hyperparameters that are involved in parameter size or constraint declarations. Then additional data is read in corresponding to predictors for new outcomes that have yet to be observed. There is no need to repeat the model or unneeded transformed parameters or generated quantities. The complete follow-on program for prediction just declares the predictors in the data, the original parameters, and then the predictions in the generated quantities block.

```
data {
  int<lower=0> N_tilde;
  vector[N_tilde] x_tilde;
}
parameters {
  real alpha;
  real beta;
  real<lower=0> sigma;
}
generated quantities {
```

```
  vector[N_tilde] y_tilde
    = normal_rng(alpha + beta * x_tilde, sigma);
}
```

When running stand-alone generated quantities, the inputs required are the original draws for the parameters and any predictors corresponding to new predictions, and the output will be draws for $\tilde{y}$ or derived quantities such as event probabilities.

Any posterior predictive quantities desired may be generated this way. For example, event probabilities are estimated in the usual way by defining indicator variables in the generated quantities block.

29. Simulation-Based Calibration

A Bayesian posterior is calibrated if the posterior intervals have appropriate coverage. For example, 80% intervals are expected to contain the true parameter 80% of the time. If data is generated according to a model, Bayesian posterior inference with respect to that model is calibrated by construction. Simulation-based calibration (SBC) exploits this property of Bayesian inference to asses the soundness of a posterior sampler. Roughly, the way it works is by simulating parameters according to the prior, then simulating data conditioned on the simulated parameters, then testing posterior calibration of the inference algorithm over independently simulated data sets. This chapter follows Talts et al. (2018), which improves on the original approach developed by Cook, Gelman, and Rubin (2006).

29.1. Bayes is calibrated by construction

Suppose a Bayesian model is given in the form of a prior density $p(\theta)$ and sampling density $p(y \mid \theta)$. Now consider a process that first simulates parameters from the prior,

$$\theta^{\text{sim}} \sim p(\theta),$$

and then simulates data given the parameters,

$$y^{\text{sim}} \sim p(y \mid \theta^{\text{sim}}).$$

By the definition of conditional densities, the simulated data and parameters constitute an independent draw from the model's joint distribution,

$$(y^{\text{sim}}, \theta^{\text{sim}}) \sim p(y, \theta).$$

From Bayes's rule, it follows that for any observed (fixed) data y,

$$p(\theta \mid y) \propto p(y, \theta).$$

Therefore, the simulated parameters constitute a draw from the posterior for the simulated data,

$$\theta^{\text{sim}} \sim p(\theta \mid y^{\text{sim}}).$$

Now consider an algorithm that produces a sequence of draws from the posterior given this simulated data,

$$\theta^{(1)}, \ldots, \theta^{(M)} \sim p(\theta \mid y^{\text{sim}}).$$

Because θ^{sim} is also distributed as a draw from the posterior, the rank statistics of θ^{sim} with respect to $\theta^{(1)}, \ldots \theta^{(M)}$ should be uniform.

This is one way to define calibration, because it follows that posterior intervals will have appropriate coverage (A. Philip Dawid 1982; Gneiting, Balabdaoui, and Raftery 2007). If the rank of θ^{sim} is uniform among the draws $\theta^{(1)}, \ldots, \theta^{(M)}$, then for any 90% interval selected, the probability the true value θ^{sim} falls in it will also be 90%. The same goes for any other posterior interval.

29.2. Simulation-based calibration

Suppose the Bayesian model to test has joint density

$$p(y, \theta) = p(y \mid \theta) \cdot p(\theta),$$

with data y and parameters θ (both are typically multivariate). Simulation-based calibration works by generating N simulated parameter and data pairs according to the joint density,

$$(y^{\text{sim}(1)}, \theta^{\text{sim}(1)}), \ldots, (y^{\text{sim}(N)}, \theta^{\text{sim}(N)}), \sim p(y, \theta).$$

For each simulated data set $y^{\text{sim}(n)}$, use the algorithm to be tested to generate M posterior draws, which if everything is working properly, will be distributed marginally as

$$\theta^{(n,1)}, \ldots, \theta^{(n,M)} \sim p(\theta \mid y^{\text{sim}(n)}).$$

For a simulation n and parameter k, the rank of the simulated parameter among the posterior draws is

$$\begin{aligned} r_{n,k} &= \text{rank}(\theta_k^{\text{sim}(n)}, (\theta^{(n,1)}, \ldots, \theta^{(n,M)})) \\ &= \sum_{m=1}^{M} \text{I}[\theta_k^{(n,m)} < \theta_k^{\text{sim}(n)}]. \end{aligned}$$

That is, the rank is the number of posterior draws $\theta_k^{(n,m)}$ that are less than the simulated draw $\theta_k^{\text{sim}(n)}$.

If the algorithm generates posterior draws according to the posterior, the ranks should be uniformly distributed from 0 to M, so that the ranks plus one are uniformly distributed from 1 to $M + 1$,

$$r_{n,k} + 1 \sim \text{categorical}\left(\frac{1}{M+1}, \ldots, \frac{1}{M+1}\right).$$

Simulation-based calibration uses this expected behavior to test the calibration of each parameter of a model on simulated data. Talts et al. (2018) suggest plotting binned counts of $r_{1:N,k}$ for different parameters k; Cook, Gelman, and Rubin (2006) automate the process with a hypothesis test for uniformity.

29.3. SBC in Stan

Running simulation-based calibration in Stan will test whether Stan's sampling algorithm can sample from the posterior associated with data generated according to the model. The data simulation and posterior fitting and rank calculation can all be done within a single Stan program. Then Stan's posterior sampler has to be run multiple times. Each run produces a rank for each parameter being assessed for uniformity. The total set of ranks can then be tested for uniformity.

Example model

For illustration, a very simple model will suffice. Suppose there are two parameters (μ, σ) with independent priors,

$$\mu \sim \text{normal}(0, 1),$$

and

$$\sigma \sim \text{lognormal}(0, 1).$$

The data $y = y_1, \ldots, y_N$ is drawn conditionally independently given the parameters,

$$y_n \sim \text{normal}(\mu, \sigma).$$

The joint prior density is thus

$$p(\mu, \sigma) = \text{normal}(\mu \mid 0, 1) \cdot \text{lognormal}(\sigma \mid 0, 1),$$

and the sampling density is

$$p(y \mid \mu, \sigma) = \prod_{n=1}^{N} \text{normal}(y_n \mid \mu, \sigma).$$

For example, suppose the following two parameter values are drawn from the prior in the first simulation,

$$(\mu^{\text{sim}(1)}, \sigma^{\text{sim}(1)}) = (1.01, 0.23).$$

Then data $y^{\text{sim}(1)} \sim p(y \mid \mu^{\text{sim}(1)}, \sigma^{\text{sim}(1)})$ is drawn according to the sampling distribution. Next, $M = 4$ draws are taken from the posterior $\mu^{(1,m)}, \sigma^{(1,m)} \sim$

$p(\mu, \sigma \mid y^{\text{sim}(1)})$,

m	$\mu^{(1,m)}$	$\sigma^{(1,m)}$
1	1.07	0.33
2	−0.32	0.14
3	−0.99	0.26
4	1.51	0.31

Then the comparisons on which ranks are based look as follows,

m	$\text{I}(\mu^{(1,m)} < \mu^{\text{sim}(1)})$	$\text{I}(\sigma^{(1,m)} < \sigma^{\text{sim}(1)})$
1	0	0
2	1	1
3	1	0
4	0	0

The ranks are the column sums, $r_{1,1} = 2$ and $r_{1,2} = 1$. Because the simulated parameters are distributed according to the posterior, these ranks should be distributed uniformly between 0 and M, the number of posterior draws.

Testing a Stan program with simulation-based calibration

To code simulation-based calibration in a Stan program, the transformed data block can be used to simulate parameters and data from the model. The parameters, transformed parameters, and model block then define the model over the simulated data. Then, in the generated quantities block, the program records an indicator for whether each parameter is less than the simulated value. As shown above, the rank is then the sum of the simulated indicator variables.

```
transformed data {
  real mu_sim = normal_rng(0, 1);
  real<lower=0> sigma_sim = lognormal_rng(0, 1);
  int<lower=0> J = 10;
  vector[J] y_sim;
  for (j in 1:J) {
    y_sim[j] = normal_rng(mu_sim, sigma_sim);
  }
}
parameters {
  real mu;
  real<lower=0> sigma;
}
model {
```

```
  mu ~ normal(0, 1);
  sigma ~ lognormal(0, 1);
  y_sim ~ normal(mu, sigma);
}
generated quantities {
  array[2] int<lower=0, upper=1> lt_sim
      = { mu < mu_sim, sigma < sigma_sim };
}
```

To avoid confusion with the number of simulated data sets used for simulation-based calibration, `J` is used for the number of simulated data points.

The model is implemented twice—once as a data generating process using random number generators in the transformed data block, then again in the parameters and model block. This duplication is a blessing and a curse. The curse is that it's more work and twice the chance for errors. The blessing is that by implementing the model twice and comparing results, the chance of there being a mistake in the model is reduced.

Pseudocode for simulation-based calibration

The entire simulation-based calibration process is as follows, where

- `p(theta)` is the prior density
- `p(y | theta)` is the sampling density
- `K` is the number of parameters
- `N` is the total number of simulated data sets and fits
- `M` is the number of posterior draws per simulated data set

```
SBC(p(theta), p(y | theta), K, N, M)
------------------------------------
for (n in 1:N) {
    // simulate parameters and data
    theta(sim(n)) ~ p(theta)
    y(sim(n)) ~ p(y | theta(sim(n)))

    // posterior draws given simulated data
    for (m in 1:M) {
        theta(n, m) ~ p(theta | y(sim(n)))
    }
    // calculate rank of sim among posterior draws
    for (k in 1:K) {
        rank(n, k) = SUM_m I(theta[k](n,m) < theta[k](sim(n)))
```

```
    }
}
// test uniformity of each parameter
for (k in 1:K) {
    test uniformity of rank(1:N, k)
}
```

The importance of thinning

The draws from the posterior are assumed to be roughly independent. If they are not, artifacts may arise in the uniformity tests due to correlation in the posterior draws. Thus it is best to think the posterior draws down to the point where the effective sample size is roughly the same as the number of thinned draws. This may require running the code a few times to judge the number of draws required to produce a target effective sample size. This operation that can be put into a loop that doubles the number of iterations until all parameters have an effective sample size of `M`, then thinning down to `M` draws.

29.4. Testing uniformity

A simple, though not very highly powered, χ^2-squared test for uniformity can be formulated by binning the ranks $0:M$ into J bins and testing that the bins all have roughly the expected number of draws in them. Many other tests for uniformity are possible. For example, Cook, Gelman, and Rubin (2006) transform the ranks using the inverse cumulative distribution function for the standard normal and then perform a test for normality. Talts et al. (2018) recommend visual inspection of the binned plots.

The bins don't need to be exactly the same size. In general, if b_j is the number of ranks that fall into bin j and e_j is the number of ranks expected to fall into bin j (which will be proportional to its size under uniformity), the test statistic is

$$X^2 = \sum_{j=1}^{J} \frac{(b_j - e_j)^2}{e_j}.$$

The terms are approximately square standard normal, so that under the null hypothesis of uniformity,

$$X^2 \sim \text{chiSquared}(J-1),$$

with the corresponding p-value given by the complementary cumulative distribution function (CCDF) of $\text{chiSquared}(J-1)$ applied to X^2. Because this test relies on the binomial being approximately normal, the traditional advice is to make sure the expected count in each bin is at least five, i.e., $e_j \geq 5$.

Indexing to simplify arithmetic

Because there are $M+1$ possible ranks, with J bins, it is easiest to have $M+1$ be divisible by J. For instance, if $J = 20$ and $M = 999$, then there are 1000 possible ranks and an expected count in each bin of $\frac{M+1}{J} = 50$.

Distributing the ranks into bins is another fiddly operation that can be done with integer arithmetic or the floor operation. Using floor, the following function determines the bin for a rank,

$$\text{bin}(r_{n,m}, M, J) = 1 + \left\lfloor \frac{r_{n,m}}{(M+1)/J} \right\rfloor .$$

For example, with $M = 999$ and $J = 20$, $(M+1)/J = 50$. The lowest rank checks out,

$$\text{bin}(0, 999, 20) = 1 + \lfloor 0/50 \rfloor = 1,$$

as does the 50th rank,

$$\text{bin}(49, 999, 20) = 1 + \lfloor 49/50 \rfloor = 1,$$

and the 51st is appropriately put in the second bin,

$$\text{bin}(50, 999, 20) = 1 + \lfloor 50/50 \rfloor = 2.$$

The highest rank also checks out, with $\text{bin}(1000, 999, 20) = 50$.

To summarize, the following pseudocode computes the b_j values for the χ^2 test or for visualization in a histogram.

```
Inputs: M draws, J bins, N parameters, ranks r[n, m]
b[1:J] = 0
for (m in 1:M) {
  ++b[1 + floor(r[n, m] * J / (M + 1))]
}
```

where the `++b[n]` notation is a common form of syntactic sugar for `b[n] = b[n] + 1`.

In general, a great deal of care must be taken in visualizing discrete data because it's easy to introduce off-by-one errors and artifacts at the edges because of the way boundaries are computed by default. That's why so much attention must be devoted to indexing and binning.

29.5. Examples of simulation-based calibration

This section will show what the results look like when the tests pass and then when they fail. The passing test will compare a normal model and normal data generating process, whereas the second will compare a normal model with a Student-t data generating process. The first will produce calibrated posteriors, the second will not.

When things go right

Consider the following simple model for a normal distribution with standard normal and lognormal priors on the location and scale parameters.

$$\begin{aligned} \mu &\sim \text{normal}(0, 1) \\ \sigma &\sim \text{lognormal}(0, 1) \\ y_{1:10} &\sim \text{normal}(\mu, \sigma). \end{aligned}$$

The Stan program for evaluating SBC for this model is

```
transformed data {
  real mu_sim = normal_rng(0, 1);
  real<lower=0> sigma_sim = lognormal_rng(0, 1);

  int<lower=0> J = 10;
  vector[J] y_sim;
  for (j in 1:J) {
    y_sim[j] = student_t_rng(4, mu_sim, sigma_sim);
  }
}
parameters {
  real mu;
  real<lower=0> sigma;
}
model {
  mu ~ normal(0, 1);
  sigma ~ lognormal(0, 1);

  y_sim ~ normal(mu, sigma);
}
generated quantities {
  array[2] int<lower=0, upper=1> I_lt_sim
      = { mu < mu_sim, sigma < sigma_sim };
}
```

After running this for enough iterations so that the effective sample size is larger than M, then thinning to M draws (here $M = 999$), the ranks are computed and binned, and then plotted.

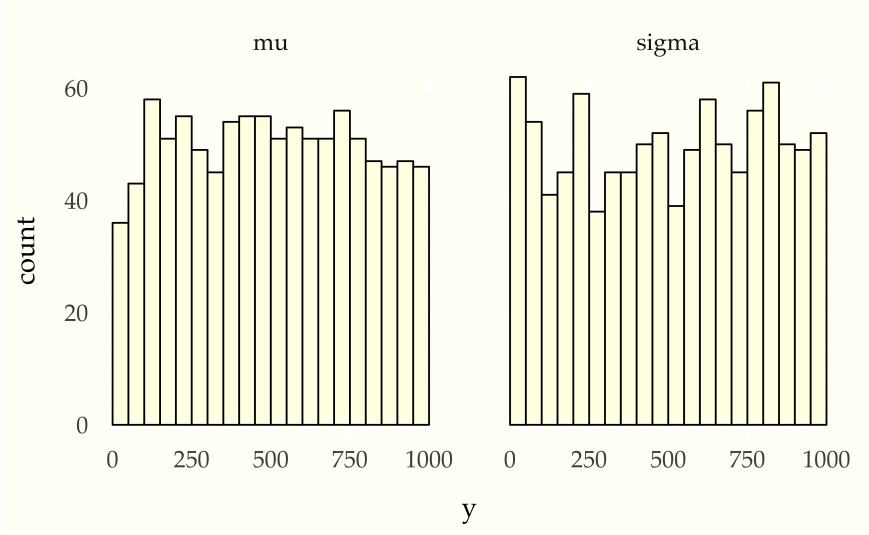


Figure 29.1: Simulation based calibration plots for location and scale of a normal model with standard normal prior on the location, standard lognormal prior on the scale. Both histograms appear uniform, which is consistent with inference being well calibrated.

When things go wrong

Now consider using a Student-t data generating process with a normal model. Compare the apparent uniformity of the well specified model with the ill-specified situation with Student-t generative process and normal model.

When Stan's sampler goes wrong

The example in the previous sections show hard-coded pathological behavior. The usual application of SBC is to diagnose problems with a sampler.

This can happen in Stan with well-specified models if the posterior geometry is too difficult (usually due to extreme stiffness that varies). A simple example is the eight schools problem, the data for which consists of sample means y_j and standard deviations σ_j of differences in test score after the same intervention in

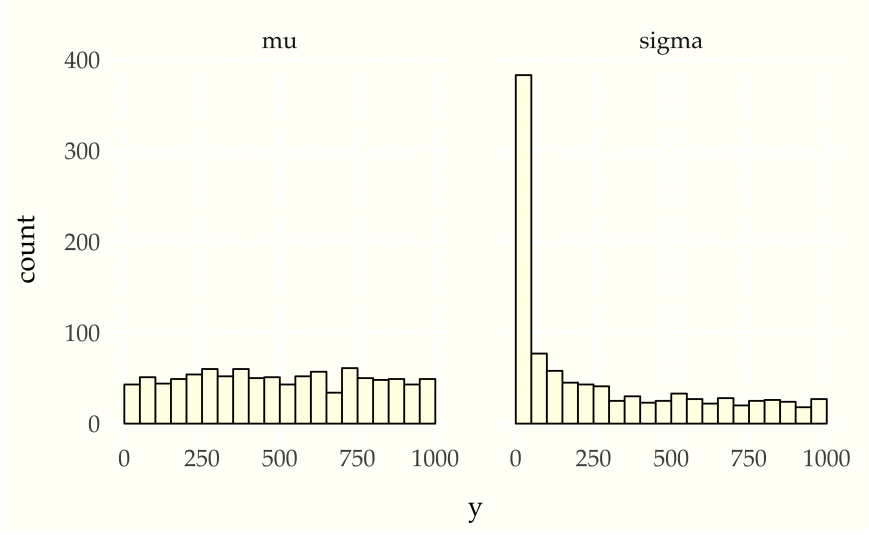


Figure 29.2: Simulation based calibration plots for location and scale of a normal model with standard normal prior on the location standard lognormal prior on the scale with mismatched generative model using a Student-t data model with 4 degrees of freedom. The mean histogram appears uniform, but the scale parameter shows simulated values much smaller than fit values, clearly signaling the lack of calibration.

$J = 8$ different schools. Donald B. Rubin (1981) applies a hierarchical model for a meta-analysis of the results, estimating the mean intervention effect and a varying effect for each school. With a standard parameterization and weak priors, this model has very challenging posterior geometry, as shown by Talts et al. (2018); this section replicates their results.

The meta-analysis model has parameters for a population mean μ and standard deviation $\tau > 0$ as well as the effect θ_j of the treatment in each school. The model has weak normal and half-normal priors for the population-level parameters,

$$\begin{aligned} \mu &\sim \text{normal}(0, 5) \\ \tau &\sim \text{normal}_+(0, 5). \end{aligned}$$

School level effects are modeled as normal given the population parameters,

$$\theta_j \sim \text{normal}(\mu, \tau).$$

The data is modeled as in a meta-analysis, given the school effect and sample standard deviation in the school,

$$y_j \sim \text{normal}(\theta_j, \sigma_j).$$

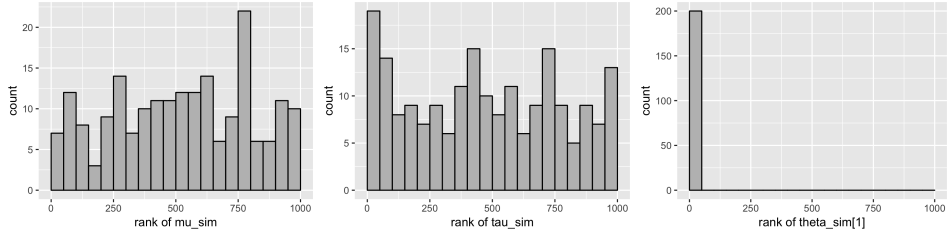
This model can be coded in Stan with a data-generating process that simulates the parameters and then simulates data according to the parameters.

```
transformed data {
  real mu_sim = normal_rng(0, 5);
  real tau_sim = abs(normal_rng(0, 5));
  int<lower=0> J = 8;
  array[J] real theta_sim = normal_rng(rep_vector(mu_sim, J), tau_sim);
  array[J] real<lower=0> sigma = abs(normal_rng(rep_vector(0, J), 5));
  array[J] real y = normal_rng(theta_sim, sigma);
}
parameters {
  real mu;
  real<lower=0> tau;
  array[J] real theta;
}
model {
  tau ~ normal(0, 5);
  mu ~ normal(0, 5);
  theta ~ normal(mu, tau);
  y ~ normal(theta, sigma);
}
generated quantities {
  int<lower=0, upper=1> mu_lt_sim = mu < mu_sim;
  int<lower=0, upper=1> tau_lt_sim = tau < tau_sim;
  int<lower=0, upper=1> theta1_lt_sim = theta[1] < theta_sim[1];
}
```

As usual for simulation-based calibration, the transformed data encodes the data-generating process using random number generators. Here, the population parameters μ and τ are first simulated, then the school-level effects θ, and then finally the observed data σ_j and y_j. The parameters and model are a direct encoding of the mathematical presentation using vectorized sampling statements. The generated quantities block includes indicators for parameter comparisons, saving only θ_1 because the schools are exchangeable in the simulation.

When fitting the model in Stan, multiple warning messages are provided that the sampler has diverged. The divergence warnings are in Stan's sampler precisely to diagnose the sampler's inability to follow the curvature in the posterior and provide independent confirmation that Stan's sampler cannot fit this model as specified.

SBC also diagnoses the problem. Here's the rank plots for running $N = 200$ simulations with 1000 warmup iterations and $M = 999$ draws per simulation used to compute the ranks.



(a) μ (b) τ (c) θ_1

Figure 29.3: Simulation based calibration plots for the eight-schools model with centered parameterization in Stan. The geometry is too difficult for the NUTS sampler to handle, as indicated by the plot for θ_1 (Figure 29.3c).

Although the population mean and standard deviation μ and τ appear well calibrated, θ_1 tells a very different story. The simulated values are much smaller than the values fit from the data. This is because Stan's no-U-turn sampler is unable to sample with the model formulated in the centered parameterization—the posterior geometry has regions of extremely high curvature as τ approaches zero and the θ_j become highly constrained. The chapter on reparameterization explains how to remedy this problem and fit this kind of hierarchical model with Stan.

30. Posterior and Prior Predictive Checks

Posterior predictive checks are a way of measuring whether a model does a good job of capturing relevant aspects of the data, such as means, standard deviations, and quantiles (Donald B. Rubin 1984; Andrew Gelman, Meng, and Stern 1996). Posterior predictive checking works by simulating new replicated data sets based on the fitted model parameters and then comparing statistics applied to the replicated data set with the same statistic applied to the original data set.

Prior predictive checks evaluate the prior the same way. Specifically, they evaluate what data sets would be consistent with the prior. They will not be calibrated with actual data, but extreme values help diagnose priors that are either too strong, too weak, poorly shaped, or poorly located.

Prior and posterior predictive checks are two cases of the general concept of predictive checks, just conditioning on different things (no data and the observed data, respectively). For hierarchical models, there are intermediate versions, as discussed in the section on hierarchical models and mixed replication.

30.1. Simulating from the posterior predictive distribution

The posterior predictive distribution is the distribution over new observations given previous observations. It's predictive in the sense that it's predicting behavior on new data that is not part of the training set. It's posterior in that everything is conditioned on observed data y.

The posterior predictive distribution for replications y^{rep} of the original data set y given model parameters θ is defined by

$$p(y^{\text{rep}} \mid y) = \int p(y^{\text{rep}} \mid \theta) \cdot p(\theta \mid y) \, \mathrm{d}\theta.$$

As with other posterior predictive quantities, generating a replicated data set y^{rep} from the posterior predictive distribution is straightforward using the generated quantities block. Consider a simple regression model with parameters $\theta = (\alpha, \beta, \sigma)$.

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

```
}
parameters {
  real alpha;
  real beta;
  real<lower=0> sigma;
}
model {
  alpha ~ normal(0, 2);
  beta ~ normal(0, 1);
  sigma ~ normal(0, 1);
  y ~ normal(alpha + beta * x, sigma);
}
```

To generate a replicated data set `y_rep` for this simple model, the following generated quantities block suffices.

```
generated quantities {
  array[N] real y_rep = normal_rng(alpha + beta * x, sigma);
}
```

The vectorized form of the normal random number generator is used with the original predictors `x` and the model parameters `alpha`, `beta`, and `sigma`. The replicated data variable `y_rep` is declared to be the same size as the original data `y`, but instead of a vector type, it is declared to be an array of reals to match the return type of the function `normal_rng`. Because the vector and real array types have the same dimensions and layout, they can be plotted against one another and otherwise compared during downstream processing.

The posterior predictive sampling for posterior predictive checks is different from usual posterior predictive sampling discussed in the chapter on posterior predictions in that the original predictors x are used. That is, the posterior predictions are for the original data.

30.2. Plotting multiples

A standard posterior predictive check would plot a histogram of each replicated data set along with the original data set and compare them by eye. For this purpose, only a few replications are needed. These should be taken by thinning a larger set of replications down to the size needed to ensure rough independence of the replications.

Here's a complete example where the model is a simple Poisson with a weakly

informative exponential prior with a mean of 10 and standard deviation of 10.

```
data {
  int<lower=0> N;
  array[N] int<lower=0> y;
}
transformed data {
  real<lower=0> mean_y = mean(to_vector(y));
  real<lower=0> sd_y = sd(to_vector(y));
}
parameters {
  real<lower=0> lambda;
}
model {
  y ~ poisson(lambda);
  lambda ~ exponential(0.2);
}
generated quantities {
  array[N] int<lower=0> y_rep = poisson_rng(rep_array(lambda, N));
  real<lower=0> mean_y_rep = mean(to_vector(y_rep));
  real<lower=0> sd_y_rep = sd(to_vector(y_rep));
  int<lower=0, upper=1> mean_gte = (mean_y_rep >= mean_y);
  int<lower=0, upper=1> sd_gte = (sd_y_rep >= sd_y);
}
```

The generated quantities block creates a variable `y_rep` for the replicated data, variables `mean_y_rep` and `sd_y_rep` for the statistics of the replicated data, and indicator variables `mean_gte` and `sd_gte` for whether the replicated statistic is greater than or equal to the statistic applied to the original data.

Now consider generating data $y \sim \text{Poisson}(5)$. The resulting small multiples plot shows the original data plotted in the upper left and eight different posterior replications plotted in the remaining boxes.

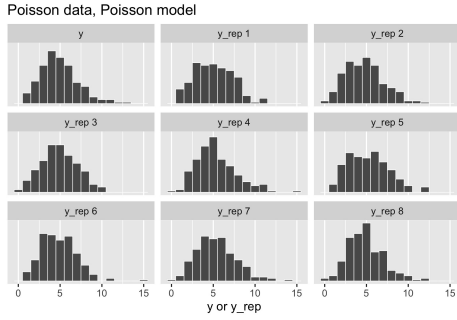


Figure 30.1: Posterior predictive checks for Poisson data generating process and Poisson model.

With a Poisson data-generating process and Poisson model, the posterior replications look similar to the original data. If it were easy to pick the original data out of the lineup, there would be a problem.

Now consider generating over-dispersed data $y \sim \text{negative-binomial2}(5, 1)$. This has the same mean as Poisson(5), namely 5, but a standard deviation of $\sqrt{5 + 5^2/1} \approx 5.5$. There is no way to fit this data with the Poisson model, because a variable distributed as Poisson(λ) has mean λ and standard deviation $\sqrt{\lambda}$, which is $\sqrt{5}$ for Poisson(5). Here's the resulting small multiples plot, again with original data in the upper left.

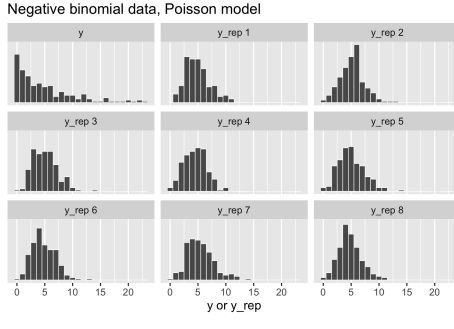


Figure 30.2: Posterior predictive checks for negative binomial data generating process and Poisson model.

This time, the original data stands out in stark contrast to the replicated data sets,

all of which are clearly more symmetric and lower variance than the original data. That is, the model's not appropriately capturing the variance of the data.

30.3. Posterior ''p-values''

If a model captures the data well, summary statistics such as sample mean and standard deviation, should have similar values in the original and replicated data sets. This can be tested by means of a p-value-like statistic, which here is just the probability the test statistic $s(\cdot)$ in a replicated data set exceeds that in the original data,

$$\Pr[s(y^{\text{rep}}) \geq s(y) \mid y] = \int \mathrm{I}\left(s(y^{\text{rep}}) \geq s(y) \mid y\right) \cdot p\left(y^{\text{rep}} \mid y\right) \, \mathrm{d}y^{\text{rep}}.$$

It is important to note that ''p-values'' is in quotes because these statistics are not classically calibrated, and thus will not in general have a uniform distribution even when the model is well specified (Bayarri and Berger 2000).

Nevertheless, values of this statistic very close to zero or one are cause for concern that the model is not fitting the data well. Unlike a visual test, this p-value-like test is easily automated for bulk model fitting.

To calculate event probabilities in Stan, it suffices to define indicator variables that take on value 1 if the event occurs and 0 if it does not. The posterior mean is then the event probability. For efficiency, indicator variables are defined in the generated quantities block.

```
generated quantities {
  int<lower=0, upper=1> mean_gt;
  int<lower=0, upper=1> sd_gt;
  {
    array[N] real y_rep = normal_rng(alpha + beta * x, sigma);
    mean_gt = mean(y_rep) > mean(y);
    sd_gt = sd(y_rep) > sd(y);
  }
}
```

The indicator variable `mean_gt` will have value 1 if the mean of the simulated data `y_rep` is greater than or equal to the mean of he original data `y`. Because the values of `y_rep` are not needed for the posterior predictive checks, the program saves output space by using a local variable for `y_rep`. The statistics `mean(u)` and `sd(y)` could also be computed in the transformed data block and saved.

For the example in the previous section, where over-dispersed data generated by a

negative binomial distribution was fit with a simple Poisson model, the following plot illustrates the posterior p-value calculation for the mean statistic.

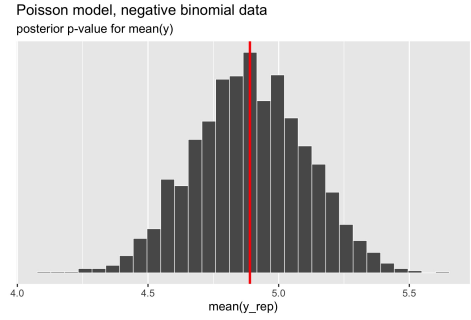


Figure 30.3: Histogram of means of replicated data sets; vertical red line at mean of original data.

The p-value for the mean is just the percentage of replicated data sets whose statistic is greater than or equal that of the original data. Using a Poisson model for negative binomial data still fits the mean well, with a posterior p-value of 0.49. In Stan terms, it is extracted as the posterior mean of the indicator variable `mean_gt`.

The standard deviation statistic tells a different story.

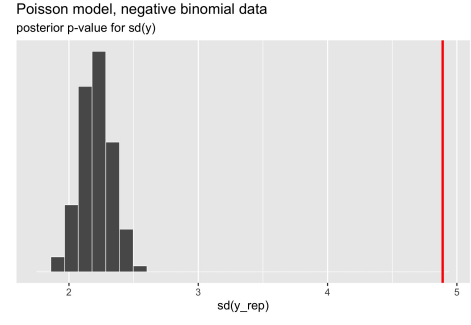


Figure 30.4: Scatterplot of standard deviations of replicated data sets; the vertical red line is at standard deviation of original data.

Here, the original data has much higher standard deviation than any of the replicated data sets. The resulting p-value estimated by Stan after a large number of

iterations is exactly zero (the absolute error bounds are fine, but a lot of iterations are required to get good relative error bounds on small p-values by sampling). In other words, there were no posterior draws in which the replicated data set had a standard deviation greater than or equal to that of the original data set. Clearly, the model is not capturing the dispersion of the original data. The point of this exercise isn't just to figure out that there's a problem with a model, but to isolate where it is. Seeing that the data is over-dispersed compared to the Poisson model would be reason to fit a more general model like the negative binomial or a latent varying effects (aka random effects) model that can account for the over-dispersion.

Which statistics to test?

Any statistic may be used for the data, but these can be guided by the quantities of interest in the model itself. Popular choices in addition to mean and standard deviation are quantiles, such as the median, 5% or 95% quantiles, or even the maximum or minimum value to test extremes.

Despite the range of choices, test statistics should ideally be ancillary, in the sense that they should be testing something other than the fit of a parameter. For example, a simple normal model of a data set will typically fit the mean and variance of the data quite well as long as the prior doesn't dominate the posterior. In contrast, a Poisson model of the same data cannot capture both the mean and the variance of a data set if they are different, so they bear checking in the Poisson case. As we saw with the Poisson case, the posterior mean for the single rate parameter was located near the data mean, not the data variance. Other distributions such as the lognormal and gamma distribution, have means and variances that are functions of two or more parameters.

30.4. Prior predictive checks

Prior predictive checks generate data according to the prior in order to asses whether a prior is appropriate (Gabry et al. 2019). A posterior predictive check generates replicated data according to the posterior predictive distribution. In contrast, the prior predictive check generates data according to the prior predictive distribution,

$$y^{\text{sim}} \sim p(y).$$

The prior predictive distribution is just like the posterior predictive distribution with no observed data, so that a prior predictive check is nothing more than the limiting case of a posterior predictive check with no data.

This is easy to carry out mechanically by simulating parameters

$$\theta^{\text{sim}} \sim p(\theta)$$

according to the priors, then simulating data

$$y^{\text{sim}} \sim p(y \mid \theta^{\text{sim}})$$

according to the data model given the simulated parameters. The result is a simulation from the joint distribution,

$$(y^{\text{sim}}, \theta^{\text{sim}}) \sim p(y, \theta)$$

and thus

$$y^{\text{sim}} \sim p(y)$$

is a simulation from the prior predictive distribution.

Coding prior predictive checks in Stan

A prior predictive check is coded just like a posterior predictive check. If a posterior predictive check has already been coded and it's possible to set the data to be empty, then no additional coding is necessary. The disadvantage to coding prior predictive checks as posterior predictive checks with no data is that Markov chain Monte Carlo will be used to sample the parameters, which is less efficient than taking independent draws using random number generation.

Prior predictive checks can be coded entirely within the generated quantities block using random number generation. The resulting draws will be independent. Predictors must be read in from the actual data set—they do not have a generative model from which to be simulated. For a Poisson regression, prior predictive sampling can be encoded as the following complete Stan program.

```
data {
  int<lower=0> N;
  vector[N] x;
}
generated quantities {
  real alpha = normal_rng(0, 1);
  real beta = normal_rng(0, 1);
  array[N] real y_sim = poisson_log_rng(alpha + beta * x);
}
```

Running this program using Stan's fixed-parameter sampler yields draws from the prior. These may be plotted to consider their appropriateness.

30.5. Example of prior predictive checks

Suppose we have a model for a football (aka soccer) league where there are J teams. Each team has a scoring rate λ_j and in each game will be assumed to

score poisson(λ_j) points. Yes, this model completely ignores defense. Suppose the modeler does not want to "put their thumb on the scale" and would rather "let the data speak for themselves" and so uses a prior with very wide tails, because it seems uninformative, such as the widely deployed

$$\lambda_j \sim \text{gamma}(\epsilon_1, \epsilon_2).$$

This is not just a manufactured example; *The BUGS Book* recommends setting $\epsilon = (0.5, 0.00001)$, which corresponds to a Jeffreys prior for a Poisson rate parameter prior (Lunn et al. 2012, 85).

Suppose the league plays a round-robin tournament wherein every team plays every other team. The following Stan model generates random team abilities and the results of such a round-robin tournament, which may be used to perform prior predictive checks.

```
data {
  int<lower=0> J;
  array[2] real<lower=0> epsilon;
}
generated quantities {
  array[J] real<lower=0> lambda;
  array[J, J] int y;
  for (j in 1:J) lambda[j] = gamma_rng(epsilon[1], epsilon[2]);
  for (i in 1:J) {
    for (j in 1:J) {
      y[i, j] = poisson_rng(lambda[i]) - poisson_rng(lambda[j]);
    }
  }
}
```

In this simulation, teams play each other twice and play themselves once. This could be made more realistic by controlling the combinatorics to only generate a single result for each pair of teams, of which there are $\binom{J}{2} = \frac{J \cdot (J-1)}{2}$.

Using the gamma$(0.5, 0.00001)$ reference prior on team abilities, the following are the first 20 simulated point differences for the match between the first two teams, $y_{1,2}^{(1:20)}$.

```
2597 -26000   5725  22496   1270   1072   4502  -2809   -302   4987
7513   7527  -3268 -12374   3828   -158 -29889   2986  -1392     66
```

That's some pretty highly scoring football games being simulated; all but one has a score differential greater than 100! In other words, this gamma$(0.5, 0.00001)$ prior is putting around 95% of its weight on score differentials above 100. Given that two teams combined rarely score 10 points, this prior is way out of line with prior knowledge about football matches; it is not only consistent with outcomes that have never occurred in the history of the sport, it puts most of the prior probability mass there.

The posterior predictive distribution can be strongly affected by the prior when there is not much observed data and substantial prior mass is concentrated around infeasible values (A. Gelman 2006).

Just as with posterior predictive distributions, any statistics of the generated data may be evaluated. Here, the focus was on score difference between a single pair of teams, but it could've been on maximums, minimums, averages, variances, etc.

In this textbook example, the prior is univariate and directly related to the expected number of points scored, and could thus be directly inspected for consistency with prior knowledge about scoring rates in football. There will not be the same kind of direct connection when the prior and data model distributions are multivariate. In these more challenging situations, prior predictive checks are an easy way to get a handle on the implications of a prior in terms of what it says the data is going to look like; for a more complex application involving spatially heterogeneous air pollution concentration, see (Gabry et al. 2019).

Prior predictive checks can also be compared with the data, but one should not expect them to be calibrated in the same way as posterior predictive checks. That would require guessing the posterior and encoding it in the prior. The goal is make sure the prior is not so wide that it will pull probability mass away from feasible values.

30.6. Mixed predictive replication for hierarchical models

Andrew Gelman, Meng, and Stern (1996) discuss the case of mixed replication for hierarchical models in which the hyperparameters remain fixed, but varying effects are replicated. This is neither a purely prior nor purely posterior predictive check, but falls somewhere in between.

For example, consider a simple varying intercept logistic regression, with intercepts α_k for $k \in 1:K$. Each data item $y_n \in \{0, 1\}$ is assumed to correspond to group $kk_n \in 1:K$. The data model is thus

$$y_n \sim \text{bernoulli}(\text{logit}^{-1}(\alpha_{kk[n]})).$$

The varying intercepts have a hierarchical normal prior,

$$\alpha_k \sim \text{normal}(\mu, \sigma).$$

The hyperparameters are themselves given weakly informative priors,

$$\begin{aligned} \mu &\sim \text{normal}(0, 2) \\ \sigma &\sim \text{lognormal}(0, 1). \end{aligned}$$

Like in a posterior predictive check, the hyperparameters μ and σ are drawn from the posterior,

$$\mu^{(m)}, \sigma^{(m)} \sim p(\mu, \sigma \mid y)$$

Like in a prior predictive check, replicated values of α are drawn from the hyperparameters,

$$\alpha_k^{\text{rep}(m)} \sim \text{normal}(\alpha_k \mid \mu^{(m)}, \sigma^{(m)}).$$

The data items are then each replicated using the replicated intercepts,

$$y_n^{\text{rep}(m)} \sim \text{bernoulli}(\text{logit}^{-1}(\alpha_{kk[n]}^{\text{rep}(m)})).$$

Thus the $y^{\text{rep}(m)}$ can be seen as a kind of posterior predictive replication of observations from new groups that were not among the original K groups.

In Stan, mixed predictive replications $y^{\text{rep}(m)}$ can be programmed directly.

```
data {
  int<lower=0> K;
  int<lower=0> N;
  array[N] int<lower=1, upper=K> kk;
  array[N] int<lower=0, upper=1> y;
}
parameters {
  real mu;
  real<lower=0> sigma;
  vector<offset=mu, multiplier=sigma>[K] alpha;
}
model {
  mu ~ normal(0, 2);                // hyperprior
  sigma ~ lognormal(0, 1);
  alpha ~ normal(mu, sigma);        // hierarchical prior
```

```
  y ~ bernoulli_logit(alpha[kk]);  // data model
}
generated quantities {
  // alpha replicated;  mu and sigma not replicated
  array[K] real alpha_rep
    = normal_rng(rep_vector(mu, K), sigma);
  array[N] int<lower=0, upper=1> y_rep
    = bernoulli_logit_rng(alpha_rep[kk]);
}
```

30.7. Joint model representation

Following Andrew Gelman, Meng, and Stern (1996), prior, posterior, and mixed replications may all be defined as posteriors from joint models over parameters and observed and replicated data.

Posterior predictive model

For example, posterior predictive replication may be formulated using distribution notation as follows.

$$\begin{aligned} \theta &\sim p(\theta) \\ y &\sim p(y \mid \theta) \\ y^{\text{rep}} &\sim p(y \mid \theta) \end{aligned}$$

The heavily overloaded distribution notation is meant to indicate that both y and y^{rep} are drawn from the same distribution, or more formally using capital letters to distinguish random variables, that the conditional densities $p_{Y^{\text{rep}}|\Theta}$ and $p_{Y|\Theta}$ are the same.

The joint density is

$$p(\theta, y, y^{\text{rep}}) = p(\theta) \cdot p(y \mid \theta) \cdot p(y^{\text{rep}} \mid \theta).$$

This again is assuming that the two distributions for y and y^{rep} are identical.

The variable y is observed, with the predictive simulation y^{rep} and parameter vector θ not observed. The posterior is $p(y^{\text{rep}}, \theta \mid y)$. Given draws from the posterior, the posterior predictive simulations y^{rep} are retained.

Prior predictive model

The prior predictive model simply drops the data component of the posterior predictive model.

$$\begin{aligned} \theta &\sim p(\theta) \\ y^{\text{rep}} &\sim p(y \mid \theta) \end{aligned}$$

This corresponds to the joint density

$$p(\theta, y^{\text{rep}}) = p(\theta) \cdot p(y^{\text{rep}} \mid \theta).$$

It is typically straightforward to draw θ from the prior and y^{rep} from the data model given θ efficiently. In cases where it is not, the model may be coded and executed just as the posterior predictive model, only with no data.

Mixed replication for hierarchical models

The mixed replication corresponds to the model

$$\begin{aligned} \phi &\sim p(\phi) \\ \alpha &\sim p(\alpha \mid \phi) \\ y &\sim p(y \mid \alpha) \\ \alpha^{\text{rep}} &\sim p(\alpha \mid \phi) \\ y^{\text{rep}} &\sim p(y \mid \phi) \end{aligned}$$

The notation here is meant to indicate that α and α^{rep} have identical distributions, as do y and y^{rep}.

This corresponds to a joint model

$$p(\phi, \alpha, \alpha^{\text{rep}}, y, y^{\text{rep}}) = p(\phi) \cdot p(\alpha \mid \phi) \cdot p(y \mid \alpha) \cdot p(\alpha^{\text{rep}} \mid \phi) \cdot p(y^{\text{rep}} \mid \alpha^{\text{rep}}),$$

where y is the only observed variable, α contains the lower-level parameters and ϕ the hyperparameters. Note that ϕ is not replicated and instead appears in the distribution for both α and α^{rep}.

The posterior is $p(\phi, \alpha, \alpha^{\text{rep}}, y^{\text{rep}} \mid y)$. From posterior draws, the posterior predictive simulations y^{rep} are kept.

31. Held-Out Evaluation and Cross-Validation

Held-out evaluation involves splitting a data set into two parts, a training data set and a test data set. The training data is used to estimate the model and the test data is used for evaluation. Held-out evaluation is commonly used to declare winners in predictive modeling competitions such as those run by Kaggle.

Cross-validation involves repeated held-out evaluations performed by partitioning a single data set in different ways. The training/test split can be done either by randomly selecting the test set, or by partitioning the data set into several equally-sized subsets and then using each subset in turn as the test data with the other folds as training data.

Held-out evaluation and cross-validation may involve any kind of predictive statistics, with common choices being the predictive log density on test data, squared error of parameter estimates, or accuracy in a classification task.

31.1. Evaluating posterior predictive densities

Given training data (x, y) consisting of parallel sequences of predictors and observations and test data $(\tilde{x}, \tilde{y})$ of the same structure, the posterior predictive density is

$$p(\tilde{y} \mid \tilde{x}, x, y) = \int p(\tilde{y} \mid \tilde{x}, \theta) \cdot p(\theta \mid x, y) \, \mathrm{d}\theta,$$

where θ is the vector of model parameters. This predictive density is the density of the test observations, conditioned on both the test predictors $\tilde{x}$ and the training data (x, y).

This integral may be calculated with Monte Carlo methods as usual,

$$p(\tilde{y} \mid \tilde{x}, x, y) \approx \frac{1}{M} \sum_{m=1}^{M} p(\tilde{y} \mid \tilde{x}, \theta^{(m)}),$$

where the $\theta^{(m)} \sim p(\theta \mid x, y)$ are draws from the posterior given only the training data (x, y).

To avoid underflow in calculations, it will be more stable to compute densities on the log scale. Taking the logarithm and pushing it through results in a stable computation,

$$\begin{aligned}
\log p(\tilde{y} \mid \tilde{x}, x, y) &\approx \log \frac{1}{M} \sum_{m=1}^{M} p(\tilde{y} \mid \tilde{x}, \theta^{(m)}), \\
&= -\log M + \log \sum_{m=1}^{M} p(\tilde{y} \mid \tilde{x}, \theta^{(m)}), \\
&= -\log M + \log \sum_{m=1}^{M} \exp(\log p(\tilde{y} \mid \tilde{x}, \theta^{(m)})) \\
&= -\log M + \text{log-sum-exp}_{m=1}^{M} \log p(\tilde{y} \mid \tilde{x}, \theta^{(m)})
\end{aligned}$$

where the log sum of exponentials function is defined so as to make the above equation hold,

$$\text{log-sum-exp}_{m=1}^{M} \mu_m = \log \sum_{m=1}^{M} \exp(\mu_m).$$

The log sum of exponentials function can be implemented so as to avoid underflow and maintain high arithmetic precision as

$$\text{log-sum-exp}_{m=1}^{M} \mu_m = \max(\mu) + \log \sum_{m=1}^{M} \exp(\mu_m - \max(\mu)).$$

Pulling the maximum out preserves all of its precision. By subtracting the maximum, the terms $\mu_m - \max(\mu) \leq 0$, and thus will not overflow.

Stan program

To evaluate the log predictive density of a model, it suffices to implement the log predictive density of the test data in the generated quantities block. The log sum of exponentials calculation must be done on the outside of Stan using the posterior draws of $\log p(\tilde{y} \mid \tilde{x}, \theta^{(m)})$.

Here is the code for evaluating the log posterior predictive density in a simple linear regression of the test data $\tilde{y}$ given predictors $\tilde{x}$ and training data (x, y).

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

```
  int<lower=0> N_tilde;
  vector[N_tilde] x_tilde;
  vector[N_tilde] y_tilde;
}
parameters {
  real alpha;
  real beta;
  real<lower=0> sigma;
}
model {
  y ~ normal(alpha + beta * x, sigma);
}
generated quantities {
  real log_p = normal_lpdf(y_tilde | alpha + beta * x_tilde, sigma);
}
```

Only the training data `x` and `y` are used in the model block. The test data `y_tilde` and test predictors `x_tilde` appear in only the generated quantities block. Thus the program is not cheating by using the test data during training. Although this model does not do so, it would be fair to use `x_tilde` in the model block—only the test observations `y_tilde` are unknown before they are predicted.

Given M posterior draws from Stan, the sequence `log_p[1:M]` will be available, so that the log posterior predictive density of the test data given training data and predictors is just `log_sum_exp(log_p) - log(M)`.

31.2. Estimation error

Parameter estimates

Estimation is usually considered for unknown parameters. If the data from which the parameters were estimated came from simulated data, the true value of the parameters may be known. If θ is the true value and $\hat{\theta}$ the estimate, then error is just the difference between the prediction and the true value,

$$\text{err} = \hat{\theta} - \theta.$$

If the estimate is larger than the true value, the error is positive, and if it's smaller, then error is negative. If an estimator's unbiased, then expected error is zero. So typically, absolute error or squared error are used, which will always have positive

expectations for an imperfect estimator. *Absolute error* is defined as

$$\text{abs-err} = \left|\hat{\theta} - \theta\right|$$

and *squared error* as

$$\text{sq-err} = \left(\hat{\theta} - \theta\right)^2.$$

Gneiting and Raftery (2007) provide a thorough overview of such scoring rules and their properties.

Bayesian posterior means minimize expected square error, whereas posterior medians minimize expected absolute error. Estimates based on modes rather than probability, such as (penalized) maximum likelihood estimates or maximum a posterior estimates, do not have these properties.

Predictive estimates

In addition to parameters, other unknown quantities may be estimated, such as the score of a football match or the effect of a medical treatment given to a subject. In these cases, square error is defined in the same way. If there are multiple exchangeable outcomes being estimated, $z_1, \ldots, z_N$, then it is common to report *mean square error* (MSE),

$$\text{mse} = \frac{1}{N} \sum_{n=1}^{N} \left(\hat{z}_n - z_n\right)^2.$$

To put the error back on the scale of the original value, the square root may be applied, resulting in what is known prosaically as *root mean square error* (RMSE),

$$\text{rmse} = \sqrt{\text{mean-sq-err}}.$$

Predictive estimates in Stan

Consider a simple linear regression model, parameters for the intercept α and slope β, along with predictors $\tilde{x}_n$. The standard Bayesian estimate is the expected value of $\tilde{y}$ given the predictors and training data,

$$\begin{aligned} \hat{\tilde{y}}_n &= \mathbb{E}[\tilde{y}_n \mid \tilde{x}_n, x, y] \\ &\approx \frac{1}{M} \sum_{m=1}^{M} \tilde{y}_n^{(m)} \end{aligned}$$

where $\tilde{y}_n^{(m)}$ is drawn from the data model

$$\tilde{y}_n^{(m)} \sim p(\tilde{y}_n \mid \tilde{x}_n, \alpha^{(m)}, \beta^{(m)}),$$

for parameters $\alpha^{(m)}$ and $\beta^{(m)}$ drawn from the posterior,

$$(\alpha^{(m)}, \beta^{(m)}) \sim p(\alpha, \beta \mid x, y).$$

In the linear regression case, two stages of simplification can be carried out, the first of which helpfully reduces the variance of the estimator. First, rather than averaging samples $\tilde{y}_n^{(m)}$, the same result is obtained by averaging linear predictions,

$$\begin{aligned} \hat{\tilde{y}}_n &= \mathbb{E}\left[\alpha + \beta \cdot \tilde{x}_n \mid \tilde{x}_n, x, y\right] \\ &\approx \frac{1}{M} \sum_{m=1}^{M} \alpha^{(m)} + \beta^{(m)} \cdot \tilde{x}_n. \end{aligned}$$

This is possible because

$$\tilde{y}_n^{(m)} \sim \text{normal}(\tilde{y}_n \mid \alpha^{(m)} + \beta^{(m)} \cdot \tilde{x}_n, \sigma^{(m)}),$$

and the normal distribution has symmetric error so that the expectation of $\tilde{y}_n^{(m)}$ is the same as $\alpha^{(m)} + \beta^{(m)} \cdot \tilde{x}_n$. Replacing the sampled quantity $\tilde{y}_n^{(m)}$ with its expectation is a general variance reduction technique for Monte Carlo estimates known as *Rao-Blackwellization* (Rao 1945; Blackwell 1947).

In the linear case, because the predictor is linear in the coefficients, the estimate can be further simplified to use the estimated coefficients,

$$\begin{aligned} \tilde{y}_n^{(m)} &\approx \frac{1}{M} \sum_{m=1}^{M} \left(\alpha^{(m)} + \beta^{(m)} \cdot \tilde{x}_n\right) \\ &= \frac{1}{M} \sum_{m=1}^{M} \alpha^{(m)} + \frac{1}{M} \sum_{m=1}^{M} (\beta^{(m)} \cdot \tilde{x}_n) \\ &= \frac{1}{M} \sum_{m=1}^{M} \alpha^{(m)} + \left(\frac{1}{M} \sum_{m=1}^{M} \beta^{(m)}\right) \cdot \tilde{x}_n \\ &= \hat{\alpha} + \hat{\beta} \cdot \tilde{x}_n. \end{aligned}$$

In Stan, only the first of the two steps (the important variance reduction step) can be coded in the object model. The linear predictor is defined in the generated quantities block.

```
data {
  int<lower=0> N_tilde;
  vector[N_tilde] x_tilde;
  // ...
}
// ...
generated quantities {
  vector[N_tilde] tilde_y = alpha + beta * x_tilde;
}
```

The posterior mean of `tilde_y` calculated by Stan is the Bayesian estimate $\hat{\tilde{y}}$. The posterior median may also be calculated and used as an estimate, though square error and the posterior mean are more commonly reported.

31.3. Cross-validation

Cross-validation involves choosing multiple subsets of a data set as the test set and using the other data as training. This can be done by partitioning the data and using each subset in turn as the test set with the remaining subsets as training data. A partition into ten subsets is common to reduce computational overhead. In the limit, when the test set is just a single item, the result is known as leave-one-out (LOO) cross-validation (Vehtari, Gelman, and Gabry 2017).

Partitioning the data and reusing the partitions is very fiddly in the indexes and may not lead to even divisions of the data. It's far easier to use random partitions, which support arbitrarily sized test/training splits and can be easily implemented in Stan. The drawback is that the variance of the resulting estimate is higher than with a balanced block partition.

Stan implementation with random folds

For the simple linear regression model, randomized cross-validation can be implemented in a single model. To randomly permute a vector in Stan, the simplest approach is the following.

```
functions {
  array[] int permutation_rng(int N) {
    array[N] int y;
    for (n in 1 : N) {
      y[n] = n;
    }
    vector[N] theta = rep_vector(1.0 / N, N);
```

```
    for (n in 1 : size(y)) {
      int i = categorical_rng(theta);
      int temp = y[n];
      y[n] = y[i];
      y[i] = temp;
    }
    return y;
  }
}
```

The name of the function must end in `_rng` because it uses other random functions internally. This will restrict its usage to the transformed data and generated quantities block. The code walks through an array of integers exchanging each item with another randomly chosen item, resulting in a uniformly drawn permutation of the integers `1:N`.[1]

The transformed data block uses the permutation RNG to generate training data and test data by taking prefixes and suffixes of the permuted data.

```
data {
  int<lower=0> N;
  vector[N] x;
  vector[N] y;
  int<lower=0, upper=N> N_test;
}
transformed data {
  int N_train = N - N_test;
  array[N] int permutation = permutation_rng(N);
  vector[N_train] x_train = x[permutation[1 : N_train]];
  vector[N_train] y_train = y[permutation[1 : N_train]];
  vector[N_test] x_test = x[permutation[N_train + 1 : N]];
  vector[N_test] y_test = y[permutation[N_train + 1 : N]];
}
```

Recall that in Stan, `permutation[1:N_train]` is an array of integers, so that `x[permutation[1 : N_train]]` is a vector defined for `i in 1:N_train` by

[1] The traditional approach is to walk through a vector and replace each item with a random element from the remaining elements, which is guaranteed to only move each item once. This was not done here as it'd require new categorical `theta` because Stan does not have a uniform discrete RNG built in.

```
x[permutation[1 : N_train]][i] = x[permutation[1:N_train][i]]
                               = x[permutation[i]]
```

Given the test/train split, the rest of the model is straightforward.

```
parameters {
  real alpha;
  real beta;
  real<lower=0> sigma;
}
model {
  y_train ~ normal(alpha + beta * x_train, sigma);
  { alpha, beta, sigma } ~ normal(0, 1);
}
generated quantities {
  vector[N] y_test_hat = normal_rng(alpha + beta * x_test, sigma);
  vector[N] err = y_test_sim - y_hat;
}
```

The prediction `y_test_hat` is defined in the generated quantities block using the general form involving all uncertainty. The posterior of this quantity corresponds to using a posterior mean estimator,

$$\begin{aligned} \hat{y}^{\text{test}} &= \mathbb{E}\left[y^{\text{test}} \mid x^{\text{test}}, x^{\text{train}} y^{\text{train}}\right] \\ &\approx \frac{1}{M} \sum_{m=1}^{M} \hat{y}^{\text{test}(m)}. \end{aligned}$$

Because the test set is constant and the expectation operator is linear, the posterior mean of `err` as defined in the Stan program will be the error of the posterior mean estimate,

$$\begin{aligned} \hat{y}^{\text{test}} - y^{\text{test}} &= \mathbb{E}\left[\hat{y}^{\text{test}} \mid x^{\text{test}}, x^{\text{train}}, y^{\text{train}}\right] - y^{\text{test}} \\ &= \mathbb{E}\left[\hat{y}^{\text{test}} - y^{\text{test}} \mid x^{\text{test}}, x^{\text{train}}, y^{\text{train}}\right] \\ &\approx \frac{1}{M} \sum_{m=1}^{M} \hat{y}^{\text{test}(m)} - y^{\text{test}}, \end{aligned}$$

where

$$\hat{y}^{\text{test}(m)} \sim p(y \mid x^{\text{test}}, x^{\text{train}}, y^{\text{train}}).$$

This just calculates error; taking absolute value or squaring will compute absolute error and mean square error. Note that the absolute value and square operation should *not* be done within the Stan program because neither is a linear function and the result of averaging squares is not the same as squaring an average in general.

Because the test set size is chosen for convenience in cross-validation, results should be presented on a per-item scale, such as average absolute error or root mean square error, not on the scale of error in the fold being evaluated.

User-defined permutations

It is straightforward to declare the variable `permutation` in the data block instead of the transformed data block and read it in as data. This allows an external program to control the blocking, allowing non-random partitions to be evaluated.

Cross-validation with structured data

Cross-validation must be done with care if the data is inherently structured. For example, in a simple natural language application, data might be structured by document. For cross-validation, one needs to cross-validate at the document level, not at the individual word level. This is related to mixed replication in posterior predictive checking, where there is a choice to simulate new elements of existing groups or generate entirely new groups.

Education testing applications are typically grouped by school district, by school, by classroom, and by demographic features of the individual students or the school as a whole. Depending on the variables of interest, different structured subsets should be evaluated. For example, the focus of interest may be on the performance of entire classrooms, so it would make sense to cross-validate at the class or school level on classroom performance.

Cross-validation with spatio-temporal data

Often data measurements have spatial or temporal properties. For example, home energy consumption varies by time of day, day of week, on holidays, by season, and by ambient temperature (e.g., a hot spell or a cold snap). Cross-validation must be tailored to the predictive goal. For example, in predicting energy consumption, the quantity of interest may be the prediction for next week's energy consumption given historical data and current weather covariates. This suggests an alternative to cross-validation, wherein individual weeks are each tested given previous data. This often allows comparing how well prediction performs with more or less historical data.

Approximate cross-validation

Vehtari, Gelman, and Gabry (2017) introduce a method that approximates the evaluation of leave-one-out cross validation inexpensively using only the data point log likelihoods from a single model fit. This method is documented and implemented in the R package loo (Gabry et al. 2019).

32. Poststratification

Stratification is a technique developed for survey sampling in which a population is partitioned into subgroups (i.e., stratified) and each group (i.e., stratum) is sampled independently. If the subgroups are more homogeneous (i.e., lower variance) than the population as a whole, this can reduce variance in the estimate of a quantity of interest at the population level.

Poststratification is a technique for adjusting a non-representative sample (i.e., a convenience sample or other observational data) for which there are demographic predictors characterizing the strata. It is carried out after a model is fit to the observed data, hence the name *post*stratification (Little 1993). Poststratification can be fruitfully combined with regression modeling (or more general parametric modeling), which provides estimates based on combinations of predictors (or general parameters) rather than raw counts in each stratum. Multilevel modeling is useful in determining how much partial pooling to apply in the regressions, leading to the popularity of the combination of multilevel regression and poststratification (MRP) (Park, Gelman, and Bafumi 2004).

32.1. Some examples

Earth science

Stratification and poststratification can be applied to many applications beyond survey sampling (Kennedy and Gelman 2019). For example, large-scale whole-earth soil-carbon models are fit with parametric models of how soil-carbon depends on features of an area such as soil composition, flora, fauna, temperature, humidity, etc. Given a model that predicts soil-carbon concentration given these features, a whole-earth model can be created by stratifying the earth into a grid of say 10km by 10km "squares" (they can't literally be square because the earth's surface is topologically a sphere). Each grid area has an estimated makeup of soil type, forestation, climate, etc. The global level of soil carbon is then estimated using poststratification by simply summing the expected soil carbon estimated for each square in the grid (Paustian et al. 1997). Dynamic models can then be constructed by layering a time-series component, varying the poststratification predictors over time, or both (Field et al. 1998).

Polling

Suppose a university's administration would like to estimate the support for a given proposal among its students. A poll is carried out in which 490 respondents are undergraduates, 112 are graduate students, and 47 are continuing education students. Now suppose that support for the issue among the poll respondents is is 25% among undergraduate students (subgroup 1), 40% among graduate students (subgroup 2), and 80% among continuing education students (subgroup 3). Now suppose that the student body is made up of 20,000 undergraduates, 5,000 graduate students, and 2,000 continuing education students. It is important that our subgroups are exclusive and exhaustive, i.e., they form a partition of the population.

The proportion of support in the poll among students in each group provides a simple maximum likelihood estimate $\theta^* = (0.25, 0.5, 0.8)$ of support in each group for a simple Bernoulli model where student n's vote is modeled as

$$y_n \sim \text{bernoulli}(\theta_{jj[n]}),$$

where $jj[n] \in 1:3$ is the subgroup to which the n-th student belongs.

An estimate of the population prevalence of support for the issue among students can be constructed by simply multiplying estimated support in each group by the size of each group. Letting $N = (20\,000, 5\,000, 2\,000)$ be the subgroup sizes, the poststratified estimate of support in the population ϕ^* is estimated by

$$\phi^* = \frac{\sum_{j=1}^{3} \theta_j^* \cdot N_j}{\sum_{j=1}^{3} N_j}.$$

Plugging in our estimates and population counts yields

$$\begin{aligned} \phi* &= \frac{0.25 \cdot 20\,000 + 0.4 \cdot 5\,000 + 0.8 \cdot 2\,000}{20\,000 + 5\,000 + 2\,000} \\ &= \frac{8\,600}{27\,000} \\ &\approx 0.32. \end{aligned}$$

32.2. Bayesian poststratification

Considering the same polling data from the previous section in a Bayesian setting, the uncertainty in the estimation of subgroup support is pushed through predic-

tive inference in order to get some idea of the uncertainty of estimated support. Continuing the example of the previous section, the data model remains the same,

$$y_n \sim \text{bernoulli}(\theta_{jj[n]}),$$

where $jj[n] \in 1:J$ is the group to which item n belongs and θ_j is the proportion of support in group j.

This can be reformulated from a Bernoulli model to a binomial model in the usual way. Letting A_j be the number of respondents in group j and a_j be the number of positive responses in group j, the data model may be reduced to the form

$$a_j \sim \text{binomial}(A_j, \theta_j).$$

A simple uniform prior on the proportion of support in each group completes the model,

$$\theta_j \sim \text{beta}(1, 1).$$

A more informative prior could be used if there is prior information available about support among the student body.

Using sampling, draws $\theta^{(m)} \sim p(\theta \mid y)$ from the posterior may be combined with the population sizes N to estimate ϕ, the proportion of support in the population,

$$\phi^{(m)} = \frac{\sum_{j=1}^{J} \theta_j^{(m)} \cdot N_j}{\sum_{j=1}^{J} N_j}.$$

The posterior draws for $\phi^{(m)}$ characterize expected support for the issue in the entire population. These draws may be used to estimate expected support (the average of the $\phi^{(m)}$), posterior intervals (quantiles of the $\phi^{(m)}$), or to plot a histogram.

32.3. Poststratification in Stan

The maximum likelihood and Bayesian estimates can be handled with the same Stan program. The model of individual votes is collapsed to a binomial, where A_j is the number of voters from group j, a_j is the number of positive responses from group j, and N_j is the size of group j in the population.

```
data {
  int<lower=1> J;
  array[J] int<lower=0> A;
  array[J] int<lower=0> a;
  vector<lower=0>[J] N;
}
parameters {
  vector<lower=0, upper=1>[J] theta;
}
model {
  a ~ binomial(A, theta);
}
generated quantities {t
  real<lower=0, upper=1> phi = dot(N, theta) / sum(N);
}
```

The binomial distribution statement is vectorized, and implicitly generates the joint likelihood for the J terms. The prior is implicitly uniform on $(0, 1)$, the support of θ. The summation is computed using a dot product and the sum function, which is why `N` was declared as a vector rather than as an array of integers.

32.4. Regression and poststratification

In applications to polling, there are often numerous demographic features like age, gender, income, education, state of residence, etc. If each of these demographic features induces a partition on the population, then their product also induces a partition on the population. Often sources such as the census have matching (or at least matchable) demographic data; otherwise it must be estimated.

The problem facing poststratification by demographic feature is that the number of strata increases exponentially as a function of the number of features. For instance, 4 age brackets, 2 sexes, 5 income brackets, and 50 states of residence leads to $5 \cdot 2 \cdot 5 \cdot 50 = 2000$ strata. Adding another 5-way distinction, say for education level, leads to 10,000 strata. A simple model like the one in the previous section that takes an independent parameter θ_j for support in each stratum is unworkable in that many groups will have zero respondents and almost all groups will have very few respondents.

A practical approach to overcoming the problem of low data size per stratum is to use a regression model. Each demographic feature will require a regression coefficient for each of its subgroups, but now the parameters add to rather than

multiply the total number of parameters. For example, with 4 age brackets, 2 sexes, 5 income brackets, and 50 states of residence, there are only $4 + 2 + 5 + 50 = 61$ regression coefficients to estimate. Now suppose that item n has demographic features $\text{age}_n \in 1:5$, $\text{sex}_n \in 1:2$, $\text{income}_n \in 1:5$, and $\text{state}_n \in 1:50$. A logistic regression may be formulated as

$$y_n \sim \text{bernoulli}(\text{logit}^{-1}(\alpha + \beta_{\text{age}[n]} + \gamma_{\text{sex}[n]} + \delta_{\text{income}[n]} + \epsilon_{\text{state}[n]})),$$

where $\text{age}[n]$ is the age of the n-th respondent, $\text{sex}[n]$ is their sex, $\text{income}[n]$ their income and $\text{state}[n]$ their state of residence. These coefficients can be assigned priors, resulting in a Bayesian regression model.

To poststratify the results, the population size for each combination of predictors must still be known. Then the population estimate is constructed as

$$\sum_{i=1}^{5} \sum_{j=1}^{2} \sum_{k=1}^{5} \sum_{m=1}^{50} \text{logit}^{-1}(\alpha + \beta_i + \gamma_j + \delta_k + \eta_m) \cdot \text{pop}_{i,j,k,m},$$

where $\text{pop}_{i,j,k,m}$ is the size of the subpopulation with age i, sex j, income level k, and state of residence m.

As formulated, it should be clear that any kind of prediction could be used as a basis for poststratification. For example, a Gaussian process or neural network could be used to produce a non-parametric model of outcomes y given predictors x.

32.5. Multilevel regression and poststratification

With large numbers of demographic features, each cell may have very few items in it with which to estimate regression coefficients. For example, even in a national-level poll of 10,000 respondents, if they are divided by the 50 states, that's only 200 respondents per state on average. When data sizes are small, parameter estimation can be stabilized and sharpened by providing hierarchical priors. With hierarchical priors, the data determines the amount of partial pooling among the groups. The only drawback is that if the number of groups is small, it can be hard to fit these models without strong hyperpriors.

The model introduced in the previous section had the data model

$$y_n \sim \text{bernoulli}(\text{logit}^{-1}(\alpha + \beta_{\text{age}[n]} + \gamma_{\text{sex}[n]} + \delta_{\text{income}[n]} + \epsilon_{\text{state}[n]})).$$

The overall intercept can be given a broad fixed prior,

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

The other regression parameters can be given hierarchical priors,

$$\begin{aligned}
\beta_{1:4} &\sim \text{normal}(0, \sigma^{\beta}) \\
\gamma_{1:2} &\sim \text{normal}(0, \sigma^{\gamma}) \\
\delta_{1:5} &\sim \text{normal}(0, \sigma^{\delta}) \\
\epsilon_{1:50} &\sim \text{normal}(0, \sigma^{\epsilon}).
\end{aligned}$$

The hyperparameters for scale of variation within a group can be given simple standard hyperpriors,

$$\sigma^{\beta}, \sigma^{\gamma}, \sigma^{\delta}, \sigma^{\epsilon} \sim \text{normal}(0, 1).$$

The scales of these fixed hyperpriors need to be determined on a problem-by-problem basis, though ideally they will be close to standard (mean zero, unit variance).

Dealing with small partitions and non-identifiability

The multilevel structure of the models used for multilevel regression and poststratification consist of a sum of intercepts that vary by demographic feature. This immediately introduces non-identifiability. A constant added to each state coefficient and subtracted from each age coefficient leads to exactly the same likelihood.

This is non-identifiability that is only mitigated by the (hierarchical) priors. When demographic partitions are small, as they are with several categories in the example, it can be more computationally tractable to enforce a sum-to-zero constraint on the coefficients. Other values than zero will by necessity be absorbed into the intercept, which is why it typically gets a broader prior even with standardized data. With a sum to zero constraint, coefficients for binary features will be negations of each other. For example, because there are only two sex categories, $\gamma_2 = -\gamma_1$.

To implement sum-to-zero constraints,

```
parameters {
  vector[K - 1] alpha_raw;
// ...
}
transformed parameters {
  vector<multiplier=sigma_alpha>[K] alpha
    = append_row(alpha_raw, -sum(alpha_raw));
// ...
}
model {
```

```
  alpha ~ normal(0, sigma_alpha);
}
```

This prior is hard to interpret in that there are `K` normal distributions, but only `K - 1` free parameters. An alternative is to put the prior only on `alpha_raw`, but that is also difficult to interpret.

Soft constraints can be more computationally tractable. They are also simpler to implement.

```
parameters {
  vector<multiplier=alpha>[K] alpha;
// ...
}
model {
  alpha ~ normal(0, sigma_alpha);
  sum(alpha) ~ normal(0, 0.001);
}
```

This leaves the regular prior, but adds a second prior that concentrates the sum near zero. The scale of the second prior will need to be established on a problem and data-set specific basis so that it doesn't shrink the estimates beyond the shrinkage of the hierarchical scale parameters.

Note that in the hierarchical model, the values of the coefficients when there are only two coefficients should be the same absolute value but opposite signs. Any other difference could be combined into the overall intercept α. Even with a wide prior on the intercept, the hyperprior on σ^γ may not be strong enough to enforce that, leading to a weak form non-identifiability in the posterior. Enforcing a (hard or soft) sum-to-zero constraint can help mitigate non-identifiability. Whatever prior is chosen, prior predictive checks can help diagnose problems with it.

None of this work to manage identifiability in multilevel regressions has anything to do with the poststratification; it's just required to fit a large multilevel regression with multiple discrete categories. Having multiple intercepts always leads to weak non-identifiability, even with the priors on the intercepts all centered at zero.

32.6. Coding MRP in Stan

Multilevel regression and poststratification can be coded directly in Stan. To code the non-centered parameterization for each coefficient, which will be required for sampling efficiency, the `multiplier` transform is used on each of the parameters.

The combination of

```
vector<multiplier=s>[K] a;
// ...
a ~ normal(0, s);
```

implements a non-centered parameterization for `a`; a centered parameterization would drop the `multiplier` specification. The prior scale `s` is being centered here. The prior location is fixed to zero in multilevel regressions because there is an overall intercept; introducing a location parameters in the prior would introduce non-identifiability with the overall intercept. The centered parameterization drops the `multiplier`.

Here is the full Stan model, which performs poststratification in the generated quantities using population sizes made available through data variable `P`.

```
data {
  int<lower=0> N;
  array[N] int<lower=1, upper=4> age;
  array[N] int<lower=1, upper=5> income;
  array[N] int<lower=1, upper=50> state;
  array[N] int<lower=0> y;
  array[4, 5, 50] int<lower=0> P;
}
parameters {
  real alpha;
  real<lower=0> sigma_beta;
  vector<multiplier=sigma_beta>[4] beta;
  real<lower=0> sigma_gamma;
  vector<multiplier=sigma_gamma>[5] gamma;
  real<lower=0> sigma_delta;
  vector<multiplier=sigma_delta>[50] delta;
}
model {
  y ~ bernoulli_logit(alpha + beta[age] + gamma[income] + delta[state]);
  alpha ~ normal(0, 2);
  beta ~ normal(0, sigma_beta);
  gamma ~ normal(0, sigma_gamma);
  delta ~ normal(0, sigma_delta);
  { sigma_beta, sigma_gamma, sigma_delta } ~ normal(0, 1);
}
```

```
generated quantities {
  real expect_pos = 0;
  int total = 0;
  for (b in 1:4) {
    for (c in 1:5) {
      for (d in 1:50) {
        total += P[b, c, d];
        expect_pos
          += P[b, c, d]
             * inv_logit(alpha + beta[b] + gamma[c] + delta[d]);
      }
    }
  }
  real<lower=0, upper=1> phi = expect_pos / total;
}
```

Unlike in posterior predictive inference aimed at uncertainty, there is no need to introduce binomial sampling uncertainty into the estimate of expected positive votes. Instead, generated quantities are computed as expectations. In general, it is more efficient to work in expectation if possible (the Rao-Blackwell theorem says it's at least as efficient to work in expectation, but in practice, it can be much much more efficient, especially for discrete quantities).

Binomial coding

In some cases, it can be more efficient to break the data down by group. Suppose there are $4 \times 5 \times 2 \times 50 = 2000$ groups. The data can be broken down into a size-2000 array, with entries corresponding to total vote counts in that group

```
int<lower=0> G;
array[G] int<lower=1, upper=4> age;
array[G] int<lower=1, upper=5> income;
array[G] int<lower=1, upper=50> state;
```

Then the number of positive votes and the number of total votes are collected into two parallel arrays indexed by group.

```
array[G] int<lower=0> pos_votes;
array[G] int<lower=0> total_votes;
```

Finally, the data model is converted to binomial.

```
pos_votes ~ binomial_logit(total_votes,
                           alpha + beta[age] + ...);
```

The predictors look the same because of the way the age and other data items are coded.

Coding binary groups

In this first model, sex is not included as a predictor. With only two categories, it needs to be modeled separately, because it is not feasible to build a hierarchical model with only two cases. A sex predictor is straightforward to add to the data block; it takes on values 1 or 2 for each of the N data points.

```
  array[N] int<lower=1, upper=2> sex;
```

Then add a single regression coefficient as a parameter,

```
  real epsilon;
```

In the log odds calculation, introduce a new term

```
[epsilon, -epsilon][sex]';
```

That is, the data model will now look like

```
  y ~ bernoulli_logit(alpha + beta[age] + gamma[income] + delta[state]
                      + [epsilon, -epsilon][sex]');
```

For data point n, the expression [epsilon, -epsilon][sex] takes on value [epsilon, -epsilon][sex][n], which with Stan's multi-indexing reduces to [epsilon, -epsilon][sex[n]]. This term evaluates to epsilon if sex[n] is 1 and to -epsilon if sex[n] is 2. The result is effectively a sum-to-zero constraint on two sex coefficients. The ' at the end transposes [epsilon, -epsilon][sex] which is a row_vector into a vector that can be added to the other variables.

Finally, a prior is needed for the coefficient in the model block,

```
epsilon ~ normal(0, 2);
```

As with other priors in multilevel models, the posterior for epsilon should be investigated to make sure it is not unrealistically wide.

32.7. Adding group-level predictors

If there are group-level predictors, such as average income in a state, or vote share in a previous election, these may be used as predictors in the regression. They

will not pose an obstacle to poststratification because they are at the group level. For example, suppose the average income level in the state is available as the data variable

```
array[50] real<lower=0> income;
```

then a regression coefficient `psi` can be added for the effect of average state income,

```
real psi;
```

with a fixed prior,

```
psi ~ normal(0, 2);
```

This prior assumes the `income` predictor has been standardized. Finally, a term is added to the regression for the fixed predictor,

```
y ~ bernoulli_logit(alpha + beta[age] + ... + delta[state]
                    + income[state] * psi);
```

And finally, the formula in the generated quantities block is also updated,

```
expect_pos
  += P[b, c, d]
     * inv_logit(alpha + beta[b] + gamma[c] + delta[d]
             + income[d] * psi);
```

Here `d` is the loop variable looping over states. This ensures that the poststratification formula matches the model formula.

33. Decision Analysis

Statistical decision analysis is about making decisions under uncertainty. In order to make decisions, outcomes must have some notion of "utility" associated with them. The so-called "Bayes optimal" decision is the one that maximizes expected utility (or equivalently, minimizes expected loss). This chapter shows how Stan can be used to simultaneously estimate the distribution of outcomes based on decisions and compute the required expected utilities.

33.1. Outline of decision analysis

Following Andrew Gelman et al. (2013), Bayesian decision analysis can be factored into the following four steps.

1. Define a set X of possible outcomes and a set D of possible decisions.
2. Define a probability distribution of outcomes conditional on decisions through a conditional density function $p(x \mid d)$ for $x \in X$ and $d \in D$.
3. Define a utility function $U : X \to \mathbb{R}$ mapping outcomes to their utility.
4. Choose action $d^* \in D$ with highest expected utility,

$$d^* = \arg\max_d \mathbb{E}[U(x) \mid d].$$

The outcomes should represent as much information as possible that is relevant to utility. In Bayesian decision analysis, the distribution of outcomes will typically be a posterior predictive distribution conditioned on observed data. There is a large literature in psychology and economics related to defining utility functions. For example, the utility of money is usually assumed to be strictly concave rather than linear (i.e., the marginal utility of getting another unit of money decreases the more money one has).

33.2. Example decision analysis

This section outlines a very simple decision analysis for a commuter deciding among modes of transportation to get to work: walk, bike share, public transportation, or cab. Suppose the commuter has been taking various modes of transportation for the previous year and the transportation conditions and costs have not changed

during that time. Over the year, such a commuter might accumulate two hundred observations of the time it takes to get to work given a choice of commute mode.

Step 1. Define decisions and outcomes

A decision consists of the choice of commute mode and the outcome is a time and cost. More formally,

- the set of decisions is $D = 1 : 4$, corresponding to the commute types walking, bicycling, public transportation, and cab, respectively, and
- the set of outcomes $X = \mathbb{R} \times \mathbb{R}_+$ contains pairs of numbers $x = (c, t)$ consisting of a cost c and time $t \geq 0$.

Step 2. Define density of outcome conditioned on decision

The density required is $p(x \mid d)$, where $d \in D$ is a decision and $x = (c, t) \in X$ is an outcome. Being a statistical decision problem, this density will the a posterior predictive distribution conditioned on previously observed outcome and decision pairs, based on a parameter model with parameters θ,

$$p(x \mid d, x^{\text{obs}}, d^{\text{obs}}) = \int p(x \mid d, \theta) \cdot p(\theta \mid x^{\text{obs}}, d^{\text{obs}}) \, \mathrm{d}\theta.$$

The observed data for a year of commutes consists of choice of the chosen commute mode d_n^{obs} and observed costs and times $x_n^{\text{obs}} = (c_n^{\text{obs}}, t_n^{\text{obs}})$ for $n \in 1 : 200$.

For simplicity, commute time t_n for trip n will be modeled as lognormal for a given choice of transportation $d_n \in 1 : 4$,

$$t_n \sim \text{lognormal}(\mu_{d[n]}, \sigma_{d[n]}).$$

To understand the notation, d_n, also written $d[n]$, is the mode of transportation used for trip n. For example if trip n was by bicycle, then $t_n \sim \text{lognormal}(\mu_2, \sigma_2)$, where μ_2 and σ_2 are the lognormal parameters for bicycling.

Simple fixed priors are used for each mode of transportation $k \in 1 : 4$,

$$\begin{aligned} \mu_k &\sim \text{normal}(0, 5) \\ \sigma_k &\sim \text{lognormal}(0, 1). \end{aligned}$$

These priors are consistent with a broad range of commute times; in a more realistic model each commute mode would have its own prior based on knowledge of the city and the time of day would be used as a covariate; here the commutes are taken to be exchangeable.

Cost is usually a constant function for public transportation, walking, and bicycling. Nevertheless, for simplicity, all costs will be modeled as lognormal,

$$c_n \sim \text{lognormal}(\nu_{d[n]}, \tau_{d[n]}).$$

Again, the priors are fixed for the modes of transportation,

$$\begin{aligned} \nu_k &\sim \text{normal}(0, 5) \\ \tau_k &\sim \text{lognormal}(0, 1). \end{aligned}$$

A more realistic approach would model cost conditional on time, because the cost of a cab depends on route chosen and the time it takes.

The full set of parameters that are marginalized in the posterior predictive distribution is

$$\theta = (\mu_{1:4}, \sigma_{1:4}, \nu_{1:4}, \tau_{1:4}).$$

Step 3. Define the utility function

For the sake of concreteness, the utility function will be assumed to be a simple function of cost and time. Further suppose the commuter values their commute time at $25 per hour and has a utility function that is linear in the commute cost and time. Then the utility function may be defined as

$$U(c, t) = -(c + 25 \cdot t)$$

The sign is negative because high cost is undesirable. A better utility function might have a step function or increasing costs for being late, different costs for different modes of transportation because of their comfort and environmental impact, and non-linearity of utility in cost.

Step 4. Maximize expected utility

At this point, all that is left is to calculate expected utility for each decision and choose the optimum. If the decisions consist of a small set of discrete choices, expected utility can be easily coded in Stan. The utility function is coded as a function, the observed data is coded as data, the model parameters coded as parameters, and the model block itself coded to follow the sampling distributions of each parameter.

```
functions {
  real U(real c, real t) {
    return -(c + 25 * t);
```

```
  }
}
data {
  int<lower=0> N;
  array[N] int<lower=1, upper=4> d;
  array[N] real c;
  array[N] real<lower=0> t;
}
parameters {
  vector[4] mu;
  vector<lower=0>[4] sigma;
  array[4] real nu;
  array[4] real<lower=0> tau;
}
model {
  mu ~ normal(0, 1);
  sigma ~ lognormal(0, 0.25);
  nu ~ normal(0, 20);
  tau ~ lognormal(0, 0.25);
  t ~ lognormal(mu[d], sigma[d]);
  c ~ lognormal(nu[d], tau[d]);
}
generated quantities {
  array[4] real util;
  for (k in 1:4) {
    util[k] = U(lognormal_rng(nu[k], tau[k]),
                lognormal_rng(mu[k], sigma[k]));
  }
}
```

The generated quantities block defines an array variable `util` where `util[k]`, which will hold the utility derived from a random commute for choice `k` generated according to the model parameters for that choice. This randomness is required to appropriately characterize the posterior predictive distribution of utility.

For simplicity in this initial formulation, all four commute options have their costs estimated, even though cost is fixed for three of the options. To deal with the fact that some costs are fixed, the costs would have to be hardcoded or read in as data, `nu` and `tau` would be declared as univariate, and the RNG for cost would only be employed when `k == 4`.

Defining the utility function for pairs of vectors would allow the random number generation in the generated quantities block to be vectorized.

All that is left is to run Stan. The posterior mean for `util[k]` is the expected utility, which written out with full conditioning, is

$$\begin{aligned}\mathbb{E}\Big[U(x) \mid d = k, d^{\text{obs}}, x^{\text{obs}}\Big] &= \int U(x) \cdot p(x \mid d = k, \theta) \cdot p(\theta \mid d^{\text{obs}}, x^{\text{obs}}) \, \mathrm{d}\theta \\ &\approx \frac{1}{M} \sum_{m=1}^{M} U(x^{(m)}),\end{aligned}$$

where

$$x^{(m)} \sim p(x \mid d = k, \theta^{(m)})$$

and

$$\theta^{(m)} \sim p(\theta \mid d^{\text{obs}}, x^{\text{obs}}).$$

In terms of Stan's execution, the random generation of $x^{(m)}$ is carried out with the `lognormal_rng` operations after $\theta^{(m)}$ is drawn from the model posterior. The average is then calculated after multiple chains are run and combined.

It only remains to make the decision `k` with highest expected utility, which will correspond to the choice with the highest posterior mean for `util[k]`. This can be read off of the `mean` column of the Stan's summary statistics or accessed programmatically through Stan's interfaces.

33.3. Continuous choices

Many choices, such as how much to invest for retirement or how long to spend at the gym are not discrete, but continuous. In these cases, the continuous choice can be coded as data in the Stan program. Then the expected utilities may be calculated. In other words, Stan can be used as a function from a choice to expected utilities. Then an external optimizer can call that function. This optimization can be difficult without gradient information. Gradients could be supplied by automatic differentiation, but Stan is not currently instrumented to calculate those derivatives.

34. The Bootstrap and Bagging

The bootstrap is a technique for approximately sampling from the error distribution for an estimator. Thus it can be used as a Monte Carlo method to estimate standard errors and confidence intervals for point estimates (Efron and Tibshirani 1986; 1994). It works by subsampling the original data and computing sample estimates from the subsample. Like other Monte Carlo methods, the bootstrap is plug-and-play, allowing great flexibility in both model choice and estimator.

Bagging is a technique for combining bootstrapped estimators for model criticism and more robust inference (Breiman 1996; Huggins and Miller 2019).

34.1. The bootstrap

Estimators

An estimator is nothing more than a function mapping a data set to one or more numbers, which are called "estimates". For example, the mean function maps a data set $y_{1,\ldots,N}$ to a number by

$$\text{mean}(y) = \frac{1}{N} \sum_{n=1}^{N} y_n,$$

and hence meets the definition of an estimator. Given the likelihood function

$$p(y \mid \mu) = \prod_{n=1}^{N} \text{normal}(y_n \mid \mu, 1),$$

the mean is the maximum likelihood estimator,

$$\text{mean}(y) = \arg\max_{\mu} p(y \mid \mu, 1)$$

A Bayesian approach to point estimation would be to add a prior and use the posterior mean or median as an estimator. Alternatively, a penalty function could be added to the likelihood so that optimization produces a penalized maximum likelihood estimate. With any of these approaches, the estimator is just a function from data to a number.

In analyzing estimators, the data set is being modeled as a random variable. It is assumed that the observed data is just one of many possible random samples

of data that may have been produced. If the data is modeled a random variable, then the estimator applied to the data is also a random variable. The simulations being done for the bootstrap are attempts to randomly sample replicated data sets and compute the random properties of the estimators using standard Monte Carlo methods.

The bootstrap in pseudocode

The bootstrap works by applying an estimator to replicated data sets. These replicates are created by subsampling the original data with replacement. The sample quantiles may then be used to estimate standard errors and confidence intervals.

The following pseudocode estimates 95% confidence intervals and standard errors for a generic estimate $\hat{\theta}$ that is a function of data y.

```
for (m in 1:M) {
  y_rep[m] <- sample_uniform(y)
  theta_hat[m] <- estimate_theta(y_rep[m])
}
std_error = sd(theta_hat)
conf_95pct = [ quantile(theta_hat, 0.025),
               quantile(theta_hat, 0.975) ]
```

The `sample_uniform` function works by independently assigning each element of `y_rep` an element of `y` drawn uniformly at random. This produces a sample *with replacement*. That is, some elements of `y` may show up more than once in `y_rep` and some may not appear at all.

34.2. Coding the bootstrap in Stan

The bootstrap procedure can be coded quite generally in Stan models. The following code illustrates a Stan model coding the likelihood for a simple linear regression. There is a parallel vector `x` of predictors in addition to outcomes `y`. To allow a single program to fit both the original data and random subsamples, the variable `resample` is set to 1 to resample and 0 to use the original data.

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

```
  simplex[N] uniform = rep_vector(1.0 / N, N);
  array[N] int<lower=1, upper=N> boot_idxs;
  for (n in 1:N) {
    boot_idxs[n] = resample ? categorical_rng(uniform) : n;
  }
}
parameters {
  real alpha;
  real beta;
  real<lower=0> sigma;
}
model {
  y[boot_idxs] ~ normal(alpha + beta * x[boot_idxs], sigma);
}
```

The model accepts data in the usual form for a linear regression as a number of observations N with a size N vector x of predictors and a size N vector of outcomes. The transformed data block generates a set of indexes into the data that is the same size as the data. This is done by independently sampling each entry of `boot_idxs` from `1:N`, using a discrete uniform distribution coded as a categorical random number generator with an equal chance for each outcome. If resampling is not done, the array `boot_idxs` is defined to be the sequence `1:N`, because `x == x[1:N]` and `y = y[1:N]`.

For example, when `resample == 1`, if $N = 4$, the value of `boot_idxs` might be `{2, 1, 1, 3}`, resulting in a bootstrap sample `{y[2], y[1], y[1], y[3]}` with the first element repeated twice and the fourth element not sampled at all.

The parameters are the usual regression coefficients for the intercept `alpha`, slope `beta`, and error scale `sigma`. The model uses the bootstrap index variable `boot_idx` to index the predictors as `x[boot_idx]` and outcomes as `y[boot_idx]`. This generates a new size-N vector whose entries are defined by `x[boot_idx][n] = x[boot_idx[n]]` and similarly for `y`. For example, if $N = 4$ and `boot_idxs = {2, 1, 1, 3}`, then `x[boot_idxs] = [x[2], x[1], x[1], x[3]]'` and `y[boot_idxs] = [y[2], y[1], y[1], y[3]]'`. The predictor and outcome vectors remain aligned, with both elements of the pair `x[1]` and `y[1]` repeated twice.

With the model defined this way, if `resample` is 1, the model is fit to a bootstrap subsample of the data. If `resample` is 0, the model is fit to the original data as given. By running the bootstrap fit multiple times, confidence intervals can be generated

from quantiles of the results.

34.3. Error statistics from the bootstrap

Running the model multiple times produces a Monte Carlo sample of estimates from multiple alternative data sets subsampled from the original data set. The error distribution is just the distribution of the bootstrap estimates minus the estimate for the original data set.

To estimate standard errors and confidence intervals for maximum likelihood estimates the Stan program is executed multiple times using optimization (which turns off Jacobian adjustments for constraints and finds maximum likelihood estimates). On the order of one hundred replicates is typically enough to get a good sense of standard error; more will be needed to accurate estimate the boundaries of a 95% confidence interval. On the other hand, given that there is inherent variance due to sampling the original data y, it is usually not worth calculating bootstrap estimates to high precision.

Standard errors

Here's the result of calculating standard errors for the linear regression model above with $N = 50$ data points, $\alpha = 1.2, \beta = -0.5$, and $\sigma = 1.5$. With a total of $M = 100$ bootstrap samples, there are 100 estimates of α, 100 of β, and 100 of σ. These are then treated like Monte Carlo draws. For example, the sample standard deviation of the draws for α provide the bootstrap estimate of the standard error in the estimate for α. Here's what it looks like for the above model with $M = 100$

```
parameter   estimate    std err
---------   --------    -------
    alpha      1.359      0.218
     beta     -0.610      0.204
    sigma      1.537      0.142
```

With the data set fixed, these estimates of standard error will display some Monte Carlo error. For example, here are the standard error estimates from five more runs holding the data the same, but allowing the subsampling to vary within Stan:

```
parameter   estimate    std err
---------   --------    -------
    alpha      1.359      0.206
    alpha      1.359      0.240
    alpha      1.359      0.234
    alpha      1.359      0.249
    alpha      1.359      0.227
```

Increasing M will reduce Monte Carlo error, but this is not usually worth the extra computation time as there is so much other uncertainty due to the original data sample y.

Confidence intervals

As usual with Monte Carlo methods, confidence intervals are estimated using quantiles of the draws. That is, if there are $M = 1000$ estimates of $\hat{\alpha}$ in different subsamples, the 2.5% quantile and 97.5% quantile pick out the boundaries of the 95% confidence interval around the estimate for the actual data set y. To get accurate 97.5% quantile estimates requires a much larger number of Monte Carlo simulations (roughly twenty times as large as needed for the median).

34.4. Bagging

When bootstrapping is carried through inference it is known as bootstrap aggregation, or *bagging*, in the machine-learning literature (Breiman 1996). In the simplest case, this involves bootstrapping the original data, fitting a model to each bootstrapped data set, then averaging the predictions. For instance, rather than using an estimate $\hat{\sigma}$ from the original data set, bootstrapped data sets $y^{\text{boot}(1)}, \ldots, y^{\text{boot}(N)}$ are generated. Each is used to generate an estimate $\hat{\sigma}^{\text{boot}(n)}$. The final estimate is

$$\hat{\sigma} = \frac{1}{N} \sum_{n=1}^{N} \hat{\sigma}^{\text{boot}(n)}.$$

The same would be done to estimate a predictive quantity $\tilde{y}$ for as yet unseen data.

$$\hat{\tilde{y}} = \frac{1}{N} \sum_{n=1}^{N} \hat{\tilde{y}}^{\text{boot}(n)}.$$

For discrete parameters, voting is used to select the outcome.

One way of viewing bagging is as a classical attempt to get something like averaging over parameter estimation uncertainty.

34.5. Bayesian bootstrap and bagging

A Bayesian estimator may be analyzed with the bootstrap in exactly the same way as a (penalized) maximum likelihood estimate. For example, the posterior mean and posterior median are two different Bayesian estimators. The bootstrap may be used estimate standard errors and confidence intervals, just as for any other estimator.

(Huggins and Miller 2019) use the bootstrap to assess model calibration and fitting in a Bayesian framework and further suggest using bagged estimators as a guard

against model misspecification. Bagged posteriors will typically have wider posterior intervals than those fit with just the original data, showing that the method is not a pure Bayesian approach to updating, and indicating it would not be calibrated if the model were well specified. The hope is that it can guard against over-certainty in a poorly specified model.

Part IV

Appendices

35. Using the Stan Compiler

Stan is used in most of our interfaces through the Stan compiler `stanc`. Since version 2.22, the Stan compiler has been implemented in OCaml and is referred to as stanc3. The binary name is still simply `stanc`, so this document uses both stanc and stanc3 interchangeably.

35.1. Command-line options for stanc3

The stanc3 compiler has the following command-line syntax:

```
> stanc (options) <model_file>
```

where `<model_file>` is a path to either a Stan model file ending in suffix `.stan` *or* a Stan functions file which ends in `.stanfunctions`.

The stanc3 options are:

- `--help` - Displays the complete list of stanc3 options, then exits.
- `--version` - Display stanc version number
- `--info` - Print information about the model, such as the type information for variables and the list of used distributions.
- `--name=<model_name>` - Specify the name of the class used for the implementation of the Stan model in the generated C++ code.
- `--o=<file_name>` - Specify a path to an output file for generated C++ code (default = .hpp) or auto-formatting output (default: no file/print to stdout)
- `--allow-undefined` - Do not throw a parser error if there is a function in the Stan program that is declared but not defined in the functions block.
- `--include_paths=<dir1,...dirN>` - Takes a comma-separated list of directories that may contain a file in an `#include` directive.
- `--use-opencl` - If set, will use additional Stan OpenCL features enabled in the Stan-to-C++ compiler.
- `--auto-format` - Pretty prints the program to the console. See more on auto formatting.

- `--canonicalize` - Make changes to the program before pretty-printing by specifying options in a comma separated list. Options are 'deprecations', 'parentheses', 'braces', 'includes'.
- `--max-line-length=<number>` - Set the column number at which formatting with `--auto-format` attempts to split lines. The default value is 78, which results in most lines being shorter than 80 characters.
- `--print-canonical` - Synonymous with `--auto-format --canonicalize=[all options]`.
- `--print-cpp` - If set, output the generated C++ Stan model class to stdout.
- `--standalone-functions` - If set, only generate the code for the functions defined in the file. This is the default behavior for `.stanfunctions` files.
- `--O0` (Default) Do not apply optimizations to the Stan code.
- `--O1` Apply level 1 compiler optimizations (only basic optimizations).
- `--Oexperimental` **WARNING:** *This is currently an experimental feature whose components are not thoroughly tested and may not improve a programs performance!* Allow the compiler to apply all optimizations to the Stan code.
- `--O` **WARNING:** *This is currently an experimental feature whose components are not thoroughly tested and may not improve a programs performance!* Same as `--Oexperimental`. Allow the compiler to apply all optimizations to the Stan code.
- `--warn-uninitialized` - Emit warnings about uninitialized variables to stderr. Currently an experimental feature.
- `--warn-pedantic` - Emit warnings in Pedantic mode which warns of potential issues in the meaning of your program.

The compiler also provides a number of debug options which are primarily of interest to stanc3 developers; use the `--help` option to see the full set.

35.2. Understanding stanc3 errors and warnings

During model compilation, stanc can produce a variety of errors (issues that prevent the model from being compiled) and warnings (non-fatal issues that should still be considered).

Warnings

Even without the optional `--warn-pedantic` and `--warn-uninitialized` command line flags, both of which enable additional warnings, stanc can still produce warnings about your program. In particular, warnings will be produced in two situations

1. A completely blank Stan program will produce the following warning message

```
Warning: Empty file 'empty.stan' detected;
    this is a valid stan model but likely unintended!
```

2. The use of any deprecated features will lead to warnings which will look as follows

```
Warning in 'deprecated.stan', line 2, column 0: Comments beginning with # are
 deprecated and this syntax will be removed in Stan 2.32.0. Use // to
 begin line comments; this can be done automatically using stanc
 --auto-format
```

   A single Stan program can produce many warnings during compilation.

Errors

Errors differ from warnings in their severity and format. In particular, errors are *fatal* and stop compilation, so at most one error is displayed per run of stanc.

There are five kinds of errors emitted by stanc3

1. File errors occur when the file passed to stanc is either missing or cannot be opened (i.e. has permissions issues). They look like

```
Error: file 'notfound.stan' not found or cannot be opened
```

2. Syntactic errors occur whenever a program violates the Stan language's syntax requirements. There are three kinds of errors within syntax errors; "lexing" errors mean that the input was unable to be read properly on the character level, "include" errors which occur when the `#include` directive fails, and "parsing" errors which result when the structure of the program is incorrect.

   - The lexing errors occur due to the use of invalid characters in a program. For example, a lexing error due to the use of `$` in a variable name will look like the following.

```
Syntax error in 'char.stan', line 2, column 6, lexing error:
-------------------------------------------------
  1:  data {
```

```
  2:     int $ome_variable;
             ^
  3:  }
--------------------------------------------------
Invalid character found.
```

- When an include directive is used, it can lead to errors if the included file is not found, or if a file includes itself (including a recursive loop of includes, such as A -> B -> A).

```
Syntax error in './incl.stan', line 1, column 0, included from
'./incl.stan', line 1, column 0, included from
'incl.stan', line 1, column 0, include error:
--------------------------------------------------
  1:  #include <incl.stan>
      ^
--------------------------------------------------
File incl.stan recursively included itself.
```

- It is much more common to see parsing errors, which tend to have more in-depth explanations of the error found. For example, if a user forgets to put a size on a type like vector, as in the following, this raises a parsing (structural) error in the compiler.

```
Syntax error in vec.stan', line 3, column 10 to column 11, parsing error
--------------------------------------------------
  1:  data {
  2:     int<lower=0> N;
  3:     vector x;
                ^
  4:  }
--------------------------------------------------
"[" expression "]" expected for vector size.
```

3. Semantic errors (also known as type errors) occur when a program is structured correctly but features an error in the type rules imposed by the language. An example of this is assigning a real value to a variable defined as an integer.

```
Semantic error in 'type.stan', line 2, column 3 to column 15:
-------------------------------------------------
  1:  transformed data {
  2:     int x = 1.5;
         ^
  3:  }
-------------------------------------------------
```

```
Ill-typed arguments supplied to assignment operator =: lhs has
type int and rhs has type real
```

4. The compiler will raise an error for use of any removed features for at least one version following their removal. The deprecation warnings mentioned above eventually turn into this kind of error to prompt the user to update their model. After the version of removal, these errors will be converted to one of the other types listed here, depending on the feature.

5. Finally, the compiler can raise an internal error. These are caused by bugs in the compiler, **not** your model, and we would appreciate it if you report them on the stanc3 repo with the error message provided. These errors usually say something like "This should never happen," and we apologize if they do.

35.3. Pedantic mode

Pedantic mode is a compilation option built into Stanc3 that warns you about potential issues in your Stan program.

For example, consider the following program.

```
data {
  int N;
  array[N] real x;
}
parameters {
  real sigma;
}
model {
  real mu;
  x ~ normal(mu, sigma);
}
```

When pedantic mode is turned on, the compiler will produce the following warnings.

```
Warning:
  The parameter sigma has no priors.
Warning at 'ped-mode-ex1.stan', line 10, column 14 to column 16:
  The variable mu may not have been assigned a value before its use.
Warning at 'ped-mode-ex1.stan', line 10, column 18 to column 23:
  A normal distribution is given parameter sigma as a scale parameter
  (argument 2), but sigma was not constrained to be strictly positive.
```

Here are the kinds of issues that pedantic mode will find (which are described in more detail in following sections):

- *Distribution usages issues.* Distribution arguments don't match the distribution specification, or some specific distribution is used in an inadvisable way.
- *Unused parameter.* A parameter is defined but doesn't contribute to target.
- *Large or small constant in a distribution.* Very large or very small constants are used as distribution arguments.
- *Control flow depends on a parameter.* Branching control flow (like if/else) depends on a parameter value .
- *Parameter has multiple tildes.* A parameter is on the left-hand side of multiple tildes.
- *Parameter has zero or multiple priors.* A parameter has zero or more than one prior distribution.
- *Variable is used before assignment.* A variable is used before being assigned a value.
- *Strict or nonsensical parameter bounds.* A parameter is given questionable bounds.
- *Nonlinear transformations.* When the left-hand side of a tilde statement (or first argument of a log probability function) contains a nonlinear transform which may require a Jacobian change of variables adjustment.

Some important limitations of pedantic mode are listed at the end of this chapter.

Distribution argument and variate constraint issues

When an argument to a built-in distribution certainly does not match that distribution's specification in the Stan Functions Reference, a warning is thrown. This primarily checks if any distribution argument's bounds at declaration, compile-time value, or subtype at declaration (e.g. `simplex`) is incompatible with the domain of the distribution. x

For example, consider the following program.

```
parameters {
  real unb_p;
  real<lower=0> pos_p;
}
model {
  1 ~ poisson(unb_p);
  1 ~ poisson(pos_p);
}
```

The parameter of `poisson` should be strictly positive, but `unb_p` is not constrained to be positive.

Pedantic mode produces the following warning.

```
Warning at 'ex-dist-args.stan', line 6, column 14 to column 19:
  A poisson distribution is given parameter unb_p as a rate parameter
  (argument 1), but unb_p was not constrained to be strictly positive.
```

Special-case distribution issues

Pedantic mode checks for some specific uses of distributions that may indicate a statistical mistake:

Uniform distributions

Any use of uniform distribution generates a warning, except when the variate parameter's declared `upper` and `lower` bounds exactly match the uniform distribution bounds. In general, assigning a parameter a uniform distribution can create non-differentiable boundary conditions and is not recommended.

For example, consider the following program.

```
parameters {
  real a;
  real<lower=0, upper=1> b;
}
model {
  a ~ uniform(0, 1);
  b ~ uniform(0, 1);
}
```

`a` is assigned a uniform distribution that doesn't match its constraints.

Pedantic mode produces the following warning.

```
Warning at 'uniform-warn.stan', line 6, column 2 to column 20:
  Parameter a is given a uniform distribution. The uniform distribution is
  not recommended, for two reasons: (a) Except when there are logical or
  physical constraints, it is very unusual for you to be sure that a
  parameter will fall inside a specified range, and (b) The infinite gradient
  induced by a uniform density can cause difficulties for Stan's sampling
  algorithm. As a consequence, we recommend soft constraints rather than hard
  constraints; for example, instead of giving an elasticity parameter a
  uniform(0, 1) distribution, try normal(0.5, 0.5).
```

(Inverse-) Gamma distributions

Gamma distributions are sometimes used as an attempt to assign an improper prior to a parameter. Pedantic mode gives a warning when the Gamma arguments indicate that this may be the case.

lkj_corr distribution

Any use of the `lkj_corr` distribution generates a warning that suggests using the Cholesky variant instead. See the LKJ correlation distribution section of the Stan Functions Reference for details.

Unused parameters

A warning is generated when a parameter is declared but does not have any effect on the program. This is determined by checking whether the value of the `target` variable depends in any way on each of the parameters.

For example, consider the following program.

```
parameters {
  real a;
  real b;
}
model {
  a ~ normal(1, 1);
}
```

`a` participates in the density function but `b` does not.

Pedantic mode produces the following warning.

```
Warning:
  The parameter b was declared but was not used in the density calculation.
```

Large or small constants in a distribution

When numbers with magnitude less than 0.1 or greater than 10 are used as arguments to a distribution, it indicates that some parameter is not scaled to unit value, so a warning is thrown. See the efficiency tuning section of the Stan User's guide for a discussion of scaling parameters.

For example, consider the following program.

```
parameters {
  real x;
  real y;
}
```

```
model {
  x ~ normal(-100, 100);
  y ~ normal(0, 1);
}
```

The constants `-100` and `100` suggest that `x` is not unit scaled.

Pedantic mode produces the following warning.

```
Warning at 'constants-warn.stan', line 6, column 14 to column 17:
  Argument -100 suggests there may be parameters that are not unit scale;
  consider rescaling with a multiplier (see manual section 22.12).
Warning at 'constants-warn.stan', line 6, column 19 to column 22:
  Argument 100 suggests there may be parameters that are not unit scale;
  consider rescaling with a multiplier (see manual section 22.12).
```

Control flow depends on a parameter

Control flow statements, such as `if`, `for` and `while` should not depend on parameters or functions of parameters to determine their branching conditions. This is likely to introduce a discontinuity into the density function. Pedantic mode generates a warning when any branching condition may depend on a parameter value.

For example, consider the following program.

```
parameters {
  real a;
}
model {
  // x depends on parameter a
  real x = a * a;

  int m;

  // the if-then-else depends on x which depends on a
  if(x > 0) {
    //now m depends on x which depends on a
    m = 1;
  } else {
    m = 2;
  }
```

```
  // for loop depends on m -> x -> a
  for (i in 0:m) {
    a ~ normal(i, 1);
  }
}
```

The `if` and `for` statements are control flow that depend (indirectly) on the value of the parameter `m`.

Pedantic mode produces the following warning.

```
Warning at 'param-dep-cf-warn.stan', line 11, column 2 to line 16, column 3:
  A control flow statement depends on parameter(s): a.
Warning at 'param-dep-cf-warn.stan', line 19, column 2 to line 21, column 3:
  A control flow statement depends on parameter(s): a.
```

Parameters with multiple tildes

A warning is generated when a parameter is found on the left-hand side of more than one ~ statements (or an equivalent `target +=` conditional density statement). This pattern is not inherently an issue, but it is unusual and may indicate a mistake.

Pedantic mode only searches for repeated statements, it will not for example generate a warning when a ~ statement is executed repeatedly inside of a loop.

For example, consider the following program.

```
data {
  real x;
}
parameters {
  real a;
  real b;
}
model {
  a ~ normal(0, 1);
  a ~ normal(x, 1);

  b ~ normal(1, 1);
}
```

Pedantic mode produces the following warning.

```
Warning at 'multi-tildes.stan', line 9, column 2 to column 19:
```

```
  The parameter a is on the left-hand side of more than one tildes
  statement.
```

Parameters with zero or multiple priors

A warning is generated when a parameter appears to have greater than or less than one prior distribution factor.

This analysis depends on a *factor graph* representation of a Stan program. A factor F that depends on a parameter P is called a *prior factor for P* if there is no path in the factor graph from F to any data variable except through P.

One limitation of this approach is that the compiler cannot distinguish between *modeled* `data` variables and other convenient uses of `data` variables such as data sizes or hyperparameters. This warning assumes that all data variables (except for `int` variables) are modeled data, which may cause extra warnings.

For example, consider the following program.

```
data {
  real x;
}
parameters {
  real a;
  real b;
  real c;
  real d;
}
model
{
  a ~ normal(0, 1); // this is a prior
  x ~ normal(a, 1); // this is not a prior, since data is involved

  b ~ normal(x, 1); // this is also not a prior, since data is involved

  // this is not a prior for c, since data is involved through b
  // but it is a prior for b, since the data is only involved through b
  c ~ normal(b, 1);

  //these are multiple priors:
  d ~ normal(0, 1);
  1 ~ normal(d, 1);
}
```

One prior is found for a and for b, while c only has a factor that touches a data variable and d has multiple priors.

Pedantic mode produces the following warning.

```
Warning:
  The parameter c has no priors.
Warning:
  The parameter d has 2 priors.
```

Variables used before assignment

A warning is generated when any variable is used before it has been assigned a value.

For example, consider the following program.

```
transformed data {
  real x;
  if (1 > 2) {
    x = 1;
  } else {
    print("oops");
  }
  print(x);
}
```

Since x is only assigned in one of the branches of the if statement, it might get to print(x) without having been assigned to.

Pedantic mode produces the following warning.

```
Warning at 'uninit-warn.stan', line 7, column 8 to column 9:
  The variable x may not have been assigned a value before its use.
```

Strict or nonsensical parameter bounds

Except when there are logical or physical constraints, it is very unusual for you to be sure that a parameter will fall inside a specified range. A warning is generated for all parameters declared with the bounds <lower=.., upper=..> except for <lower=0, upper=1> or <lower=-1, upper=1>.

In addition, a warning is generated when a parameter bound is found to have lower >= upper.

For example, consider the following program.

```
parameters {
  real<lower=0, upper=1> a;
  real<lower=-1, upper=1> b;
  real<lower=-2, upper=1012> c;
}
model {
  c ~ normal(b, a);
}
```

Pedantic mode produces the following warning.

```
Warning:
  Your Stan program has a parameter c with a lower and upper bound in its
  declaration. These hard constraints are not recommended, for two reasons:
  (a) Except when there are logical or physical constraints, it is very
  unusual for you to be sure that a parameter will fall inside a specified
  range, and (b) The infinite gradient induced by a hard constraint can cause
  difficulties for Stan's sampling algorithm. As a consequence, we recommend
  soft constraints rather than hard constraints; for example, instead of
  constraining an elasticity parameter to fall between 0, and 1, leave it
  unconstrained and give it a normal(0.5, 0.5) prior distribution.
```

Nonlinear transformations

When a parameter is transformed in a non-linear fashion, an adjustment must be applied to account for distortion caused by the transform. This is discussed in depth in the Changes of variables section.

This portion of pedantic mode tries to detect instances where such an adjustment would be necessary and remind the user.

For example, consider the following program.

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

Pedantic mode produces the following warning.

```
Warning:
    Left-hand side of distribution statement (~) may contain a non-linear
```

```
transform of a parameter or local variable. If it does, you need
to include a target += statement with the log absolute determinant
of the Jacobian of the transform. You could also consider defining
a transformed parameter and using jacobian += in the transformed
parameters block.
```

Pedantic mode limitations

- Constant values are sometimes uncomputable

  Pedantic mode attempts to evaluate expressions down to literal values so that they can be used to generate warnings. For example, in the code `normal(x, 1 - 2)`, the expression `1 - 2` will be evaluated to `-1`, which is not a valid variance argument so a warning is generated. However, this strategy is limited; it is often impossible to fully evaluate expressions in finite time.

- Container types

  Currently, indexed variables are not handled intelligently, so they are treated as monolithic variables. Each analysis treats indexed variables conservatively (erring toward generating fewer warnings).

- Data variables

  The declaration information for `data` variables is currently not considered, so using `data` as incompatible arguments to distributions may not generate the appropriate warnings.

- Control flow dependent on parameters in nested functions

  If a parameter is passed as an argument to a user-defined function within another user-defined function, and then some control flow depends on that argument, the appropriate warning will not be thrown.

35.4. Automatic updating and formatting of Stan programs

In addition to compiling Stan programs, stanc3 features several flags which can be used to format Stan programs and update them to the most recent Stan syntax by removing any deprecation features which can be automatically replaced.

These flags work for both `.stan` model files and `.stanfunctions` function files. They can be combined with `--o` to redirect the formatted output to a new file.

Automatic formatting

Invoking `stanc --auto-format <model_file>` will print a version of your model which has been re-formatted. The goal is to have this automatic formatting stay as

close as possible to the Stan Program Style Guide. This means spacing, indentation, and line length are all regularized. *Some* deprecated features, like the use of `#` for line comments, are replaced, but the goal is mainly to preserve the program while formatting it.

By default, this will try to split lines at or before column 78. This number can be changed using `--max-line-length`.

Canonicalizing

In addition to automatic formatting, stanc can also "canonicalize" programs by updating deprecated syntax, removing unnecessary parenthesis, and adding braces around bodies of `if` statements and `for` and `while` loops.

This can be done by using `stanc --auto-format --canonicalize=...` where `...` is a comma-separated list of options. Currently these options are:

- `deprecations`

  Removes deprecated syntax such as replacing deprecated functions with their drop-in replacements.

- `parentheses`

  Removes unnecessary extra parentheses, such as converting `y = ((x-1))` to `y = x - 1`

- `braces`

  Places braces around all blocks. For example, the following statement

  ```
  if (cond)
    //result
  ```

  will be formatted as

  ```
  if (cond) {
    //result
  }
  ```

  and similarly for both kinds of loops containing a single statement.

- `includes`

  This will pretty-print code from other files included with `#include` as part of the program. This was the default behavior prior to Stan 2.29. When not

enabled, the pretty-printer output will include the same `#include` directives as the input program.

Invoking `stanc --print-canonical <model_file>` is synonymous with running `stanc --auto-format --canonicalize=deprecations,braces,parentheses,includes`

Known issues

The formatting and canonicalizing features of stanc3 are still under development. The following are some known issues one should be aware of before using either:

- Oddly placed comments

  If your Stan program features comments in unexpected places, such as inside an expression, they may be moved in the process of formatting. Moved comments are prefixed with the string `^^^:` to indicate they originally appeared higher in the program.

  We hope to improve this functionality in future versions. For now, this can usually be avoided by manually moving the comment outside of an expression, either by placing it on its own line or following a separator such as a comma or keyword.

- Failure to recreate strange `#include` structure

  Printing *without* include inlining (`--canonicalize=includes`) can fail when includes were used in atypical locations, such as in the middle of statements. We recommend either printing with inlining enabled or reconsidering the use of includes in this way.

35.5. Optimization

The stanc3 compiler can optimize the code of Stan model during compilation. The optimized model code behaves the same as unoptimized code, but it may be faster, more memory efficient, or more numerically stable.

This section introduces the available optimization options and describes their effect.

To print out a representation of the optimized Stan program, use the stanc3 command-line flag `--debug-optimized-mir-pretty`. To print an analogous representation of the Stan program prior to optimization, use the flag `--debug-transformed-mir-pretty`.

Optimization levels

To turn optimizations on, the user specifies the desired optimization *level*. The level specifies the set of optimizations to use. The chosen optimizations are used in a specific order, with some of them applied repeatedly.

Optimization levels are specified by the numbers 0 and 1 and the 'experimental' tag:

- **O0** No optimizations are applied.
- **O1** Optimizations that are simple, do not dramatically change the program, and are unlikely to noticeably slow down compile times are applied.
- **Oexperimental** All optimizations are applied. Some of these are not thorougly tested and may not always improve a programs performance.

O0 is the default setting.

The levels include these optimizations:

- **O0** includes no optimizations.
- **O1** includes:
  - Dead code elimination
  - Copy propagation
  - Constant propagation
  - Partial evaluation
  - Function inlining
  - Matrix memory layout optimization
- **Oexperimental** includes optimizations specified by **O1** and also:
  - Automatic-differentiation level optimization
  - One step loop unrolling
  - Expression propagation
  - Lazy code motion
  - Static loop unrolling

In addition, **Oexperimental** will apply more repetitions of the optimizations, which may increase compile times.

O1 Optimizations

Dead code elimination

Dead code is code that does not affect the behavior of the program. Code is not dead if it affects `target`, the value of any outside-observable variable like transformed parameters or generated quantities, or side effects such as print statements. Removing dead code can speed up a program by avoiding unnecessary computations.

Example Stan program:

```
model {
  int i;
  i = 5;
  for (j in 1:10);
  if (0) {
    print("Dead code");
  } else {
    print("Hi!");
  }
}
```

Compiler representation of program **before dead code elimination** (simplified from the output of `--debug-transformed-mir-pretty`):

```
log_prob {
  int i = 5;
  for(j in 1:10) {
    ;
  }
  if(0) {
    FnPrint__("Dead code");
  } else {
    FnPrint__("Hi!");
  }
}
```

Compiler representation of program **after dead code elimination** (simplified from the output of `--debug-optimized-mir-pretty`):

```
log_prob {
  int i;
  FnPrint__("Hi!");
}
```

Constant propagation

Constant propagation replaces uses of a variable which is known to have a constant value `C` with that constant `C`. This removes the overhead of looking up the variable, and also makes many other optimizations possible (such as static loop unrolling and partial evaluation).

Example Stan program:

```
transformed data {
  int n = 100;
  int a[n];
  for (i in 1:n) {
    a[i] = i;
  }
}
```

Compiler representation of program **before constant propagation** (simplified from the output of `--debug-transformed-mir-pretty`):

```
prepare_data {
  data int n = 100;
  data array[int, n] a;
  for(i in 1:n) {
    a[i] = i;
  }
}
```

Compiler representation of program **after constant propagation** (simplified from the output of `--debug-optimized-mir-pretty`):

```
prepare_data {
  data int n = 100;
  data array[int, 100] a;
  for(i in 1:100) {
    a[i] = i;
  }
}
```

Copy propagation

Copy propagation is similar to expression propagation, but only propagates variables rather than arbitrary expressions. This can reduce the complexity of the code for other optimizations such as expression propagation.

Example Stan program:

```
model {
  int i = 1;
  int j = i;
  int k = i + j;
}
```

Compiler representation of program **before copy propagation** (simplified from the

output of `--debug-transformed-mir-pretty`):

```
log_prob {
    int i = 1;
    int j = i;
    int k = (i + j);
}
```

Compiler representation of program **after copy propagation** (simplified from the output of `--debug-optimized-mir-pretty`):

```
log_prob {
  int i = 1;
  int j = i;
  int k = (i + i);
}
```

Partial evaluation

Partial evaluation searches for expressions that we can replace with a faster, simpler, more memory efficient, or more numerically stable expression with the same meaning.

Example Stan program:

```
model {
  real a = 1 + 1;
  real b = log(1 - a);
  real c = a + b * 5;
}
```

Compiler representation of program **before partial evaluation** (simplified from the output of `--debug-transformed-mir-pretty`):

```
log_prob {
  real a = (1 + 1);
  real b = log((1 - a));
  real c = (a + (b * 5));
}
```

Compiler representation of program **after partial evaluation** (simplified from the output of `--debug-optimized-mir-pretty`):

```
log_prob {
  real a = 2;
  real b = log1m(a);
  real c = fma(b, 5, a);
```

```
}
```

Function inlining

Function inlining replaces each function call to each user-defined function `f` with the body of `f`. It does this by copying the function body to the call site and doing appropriately renaming the argument variables. This optimization can speed up a program by avoiding the overhead of a function call and providing more opportunities for further optimizations (such as partial evaluation).

Example Stan program:

```
functions {
  int incr(int x) {
    int y = 1;
    return x + y;
  }
}
transformed data {
  int a = 2;
  int b = incr(a);
}
```

Compiler representation of program **before function inlining** (simplified from the output of `--debug-transformed-mir-pretty`):

```
functions {
  int incr(int x) {
    int y = 1;
    return (x + y);
  }
}

prepare_data {
  data int a = 2;
  data int b = incr(a);
}
```

Compiler representation of program **after function inlining** (simplified from the output of `--debug-optimized-mir-pretty`):

```
prepare_data {
  data int a;
  a = 2;
  data int b;
```

```
  data int inline_sym1__;
  data int inline_sym3__;
  inline_sym3__ = 0;
  for(inline_sym4__ in 1:1) {
    int inline_sym2__;
    inline_sym2__ = 1;
    inline_sym3__ = 1;
    inline_sym1__ = (a + inline_sym2__);
    break;
  }
  b = inline_sym1__;
}
```

In this code, the `for` loop and `break` is used to simulate the behavior of a `return` statement. The value to be returned is held in `inline_sym1__`. The flag variable `inline_sym3__` indicates whether a return has occurred and is necessary to handle `return` statements nested inside loops within the function body.

Matrix memory layout optimization

Matrices and vector variables which require automatic-differentiation (AD) in Stan can be represented in two different forms.

The first (and default) representation is the "Array of Structs" (AoS) or "Matrix of vars" (matvar) layout. A "var" is the term used in the Stan implementation of autodiff for a single real. It is represented as a structure containing it's value and its adjoint. The AoS representation constructs matrices and vectors by simply using those structures as the elements of the matrix internally. This is flexible and very general, but many operations want to deal with the values or the adjoints as blocks, requiring expensive memory access patterns.

The second representation is the "Struct of Arrays" (SoA) or "Var of matrices" (varmat) layout. Rather than a matrix containing tiny structures of one value and one adjoint each, this representation uses a single structure which contains separately a matrix of values and a matrix of adjoints. Some operations, like iterating over elements or assigning to specific indices, become more expensive, but many matrix operations like multiplications become much faster in this representation.

More general reading on AoS vs SoA can be found on Wikipedia

This optimization pass attempts to identify which matrix or vector variables in the Stan program are candidates for using the SoA representation. The conditions change over time, but broadly speaking:

- Any Stan Math Library functions the matrix is passed to must be able to

support it.
- The matrix should not be accessed/assigned elementwise in a loop.

The debug flag `--debug-mem-patterns` will list each variable and whether it is using the AoS representation or the SoA representation.

0experimental Optimizations

Automatic-differentiation level optimization

Stan variables can have two auto-differentiation (AD) *levels*: AD or non-AD. AD variables carry gradient information with them, which allows Stan to calculate the log-density gradient, but they also have more overhead than non-AD variables. It is therefore inefficient for a variable to be AD unnecessarily. AD-level optimization sets every variable to be a floating point type unless its gradient is necessary.

Example Stan program:

```
data {
  real y;
}
model {
  real x = y + 1;
}
```

Compiler representation of program **before AD-level optimization** (simplified from the output of `--debug-transformed-mir-pretty`):

```
input_vars {
  real y;
}

log_prob {
  real x = (y + 1);
}
```

Compiler representation of program **after AD-level optimization** (simplified from the output of `--debug-optimized-mir-pretty`):

```
input_vars {
  real y;
}

log_prob {
  data real x = (y + 1);
}
```

One step loop unrolling

One step loop unrolling is similar to static loop unrolling. However, this optimization only 'unrolls' the first loop iteration, and can therefore work even when the total number of iterations is not predictable. This can speed up a program by providing more opportunities for further optimizations such as partial evaluation and lazy code motion.

Example Stan program:

```
data {
  int n;
}
transformed data {
  int x = 0;
  for (i in 1:n) {
    x += i;
  }
}
```

Compiler representation of program **before one step static loop unrolling** (simplified from the output of `--debug-transformed-mir-pretty`):

```
prepare_data {
  data int n = FnReadData__("n")[1];
  data int x = 0;
  for(i in 1:n) {
    x = (x + i);
  }
}
```

Compiler representation of program **after one step static loop unrolling** (simplified from the output of `--debug-optimized-mir-pretty`):

```
prepare_data {
  data int n = FnReadData__("n")[1];
  int x = 0;
  if((n >= 1)) {
    x = (x + 1);
    for(i in (1 + 1):n) {
      x = (x + i);
    }
  }
}
```

Expression propagation

Constant propagation replaces the uses of a variable which is known to have a constant value `E` with that constant `E`. This often results in recalculating the expression, but provides more opportunities for further optimizations such as partial evaluation. Expression propagation is always followed by lazy code motion to avoid unnecessarily recomputing expressions.

Example Stan program:

```
data {
  int m;
}
transformed data {
  int n = m+1;
  int a[n];
  for (i in 1:n-1) {
    a[i] = i;
  }
}
```

Compiler representation of program **before expression propagation** (simplified from the output of `--debug-transformed-mir-pretty`):

```
prepare_data {
  data int m = FnReadData__("m")[1];
  data int n = (m + 1);
  data array[int, n] a;
  for(i in 1:(n - 1)) {
    a[i] = i;
  }
}
```

Compiler representation of program **after expression propagation** (simplified from the output of `--debug-optimized-mir-pretty`):

```
prepare_data {
  data int m = FnReadData__("m")[1];
  data int n = (m + 1);
  data array[int, (m + 1)] a;
  for(i in 1:((m + 1) - 1)) {
    a[i] = i;
  }
}
```

Lazy code motion

Lazy code motion rearranges the statements and expressions in a program with the goals of:

- Avoiding computing expressions more than once, and
- Computing expressions as late as possible (to minimize the strain on the working memory set).

To accomplish these goals, lazy code motion will perform optimizations such as:

- Moving a repeatedly calculated expression to its own variable (also referred to as *common-subexpression elimination*)
- Moving an expression outside of a loop if it does not need to be in the loop (also referred to as *loop-invariant code motion*)

Lazy code motion can make some programs significantly more efficient by avoiding redundant or early computations.

As currently implemented in the compiler, it may move items between blocks in a way that actually increases overall computation. Improving this is an ongoing project.

Example Stan program:

```
model {
  real x;
  real y;
  real z;

  for (i in 1:10) {
    x = sqrt(10);
    y = sqrt(i);
  }
  z = sqrt(10);
}
```

Compiler representation of program before lazy code motion (simplified from the output of `--debug-transformed-mir-pretty`):

```
log_prob {
  real x;
  real y;
  real z;
  for(i in 1:10) {
```

```
    x = sqrt(10);
    y = sqrt(i);
  }
  z = sqrt(10);
}
```

Compiler representation of program after lazy code motion (simplified from the output of `--debug-optimized-mir-pretty`):

```
log_prob {
  data real lcm_sym4__;
  data real lcm_sym3__;
  real x;
  real y;
  lcm_sym4__ = sqrt(10);
  real z;
  for(i in 1:10) {
    x = lcm_sym4__;
    y = sqrt(i);
  }
  z = lcm_sym4__;
}
```

Static loop unrolling

Static loop unrolling takes a loop with a predictable number of iterations `X` and replaces it by writing out the loop body `X` times. The loop index in each repeat is replaced with the appropriate constant. This optimization can speed up a program by avoiding the overhead of a loop and providing more opportunities for further optimizations (such as partial evaluation).

Example Stan program:

```
transformed data {
  int x = 0;
  for (i in 1:4) {
    x += i;
  }
}
```

Compiler representation of program **before static loop unrolling** (simplified from the output of `--debug-transformed-mir-pretty`):

```
prepare_data {
  data int x = 0;
```

```
  for(i in 1:4) {
    x = (x + i);
  }
}
```

Compiler representation of program **after static loop unrolling** (simplified from the output of `--debug-optimized-mir-pretty`):

```
prepare_data {
  data int x;
  x = 0;
  x = (x + 1);
  x = (x + 2);
  x = (x + 3);
  x = (x + 4);
}
```

36. 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. This is the style used in the Stan documentation, and should align with the auto-formatting ability of `stanc3`.

36.1. 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.

36.2. 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.

36.3. File extensions

The recommended file extension for Stan model files is `.stan`. Files which contain only function definitions (intended for use with `#include`) should be given the `.stanfunctions` extension. A `.stanfunctions` file only includes the function definition and does not require the `functions{}` block wrapped around the function. A simple example of usage where the function is defined and saved in the file `foo.stanfunctions`:

```
real foo(real x, real y) {
  return sqrt(x * log(y));
}
```

[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 function `foo` can be accessed in the Stan program by including the path to the `foo.stanfunctions` file as:

```
functions {
  #include foo.stanfunctions;
}
// ...body...
```

For Stan data dump files, the recommended extension is `.R`, or more informatively, `.data.R`. For JSON output, the recommended extension is `.json`.

36.4. 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`. An exception is often made for terms appearing in mathematical expressions with standard names, like `A` for a matrix.

Another exception to the lowercasing recommendation, which 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] = ...
  }
}
```

36.5. 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 {
  array[N] real mu;
  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.

36.6. Parentheses and brackets

Braces for single-statement blocks

Single-statement blocks can be rendered in several ways. The preferred style is fully bracketed with the statement appearing on its own line, as follows.

```
for (n in 1:N) {
  y[n] ~ normal(mu,1);
}
```

The use of loops and conditionals 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);
```

Therefore, one should prefer to use braces. The only exception is when nesting if-else clauses, where the `else` branch contains exactly one conditional. Then, it is preferred to place the following `if` on the same line, as in the following.

```
if (x) {
  // ...
} else if (y) {
  // ...
} else {
  // ...
}
```

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 with the opening brace appearing at the end of a line.

```
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;
  // ...
}
```

The exception to this rule is local blocks which only exist for scoping reasons. The opening brace of these blocks is not associated with any control flow or block structure, so it should appear on its own line.

36.7. Conditionals

While Stan supports the full C++-style conditional syntax, allowing real or integer values to act as conditions, real values should be avoided. For a real-valued `x`, one should use

```
if (x != 0) { ...
```

in place of

```
if (x) { ...
```

Beyond stylistic choices, one should be careful using real values in a conditional expression, as direct comparison can have unexpected results due to numerical accuracy.

36.8. 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.

36.9. 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

Each statement of a program should appear on its own line. Declaring multiple variables of the same type can be accomplished in a single statement with the syntax

```
real mu, sigma;
```

No tabs

Stan programs should not contain tab characters. 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.

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 should not be space between a function name and the arguments 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`.

Unary operators are written without a space, such as in `-x`, `!y`.

No spaces in type constraints

Another exception to the above rule is when the assignment operator (=) is used inside a type constraint, such as

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

Spaces should still be used in arithmetic and following commas, as in

```
real<lower=0, upper=a * x + b> x;
```

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 a term above to indicate scoping. For example, use the following form

```
vector[J] p_distance = Phi((distance_tolerance - overshot)
                           ./ ((x + overshot) * sigma_distance))
                       - Phi((-overshot)
                             ./ ((x + overshot) * sigma_distance));
```

Here, the elementwise division operator (`./`) is aligned to clearly signal the division is occurring inside the parethenesis, while the subtraction indicates it is between the function applications (`Phi`).

For functions with multiple 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));
```

Spaces after commas

Commas should always be followed by spaces, including in function arguments, sequence literals, between variable declarations, etc.

For example,

```
normal(alpha * x[n] + beta, sigma);
```

is preferred over

[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.

```
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.

37. 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.

37.1. 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.

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

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

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.

37.2. 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`
...	...

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);
```

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

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

model might have a vector of `K` parameters for each of `J` groups; this can be declared using

```
array[J] vector[K] theta;
```

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

```
array[J, K] real theta;
```

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;
    }
}
```

37.3. 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 distribution 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 distribution 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, $\text{normal}(y \mid 0,1) * \text{normal}(y \mid 2,3)$, 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, $\text{normal}(x \mid 0.8y, \sigma) \times \text{normal}(y \mid 0.8x, \sigma)$, 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 Andrew Gelman et al. (2013) and Andrew 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.

37.4. 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.

37.5. 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_

Aguilar, Omar, and Mike West. 2000. "Bayesian Dynamic Factor Models and Portfolio Allocation." *Journal of Business & Economic Statistics* 18 (3): 338–57.

Ahnert, Karsten, and Mario Mulansky. 2011. "Odeint—Solving Ordinary Differential Equations in C++." *arXiv* 1110.3397.

Ascher, Uri M., and Linda R. Petzold. 1998. *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations*. Philadelphia: SIAM: Society for Industrial; Applied Mathematics.

Bayarri, MJ, and James O Berger. 2000. "P Values for Composite Null Models." *Journal of the American Statistical Association* 95 (452): 1127–42.

Betancourt, Michael, and Mark Girolami. 2013. "Hamiltonian Monte Carlo for Hierarchical Models." *arXiv* 1312.0906. http://arxiv.org/abs/1312.0906.

Blackwell, David. 1947. "Conditional Expectation and Unbiased Sequential Estimation." *The Annals of Mathematical Statistics* 18 (1): 105–10. https://doi.org/10.1214/aoms/1177730497.

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.

Breiman, Leo. 1996. "Bagging Predictors." *Machine Learning* 24 (2): 123–40.

Brenan, K. E., S. L. Campbell, and L. R. Petzold. 1995. *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations*. SIAM Classics in Applied Mathematics 14. Philadelphia: Society for Industrial; Applied Mathematics.

Breslow, N. E. 1975. "Analysis of Survival Data Under the Proportional Hazards Model." *International Statisticas Review* 43 (1): 45–58.

Cash, J. R., and Alan H. Karp. 1990. "A Variable Order Runge-Kutta Method for Initial Value Problems with Rapidly Varying Right-Hand Sides." *ACM Transactions on Mathematical Software* 16 (3): 201–22. https://doi.org/10.1145/79505.79507.

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.

Cook, Samantha R., Andrew Gelman, and Donald B Rubin. 2006. "Validation of Software for Bayesian Models Using Posterior Quantiles." *Journal of Computational and Graphical Statistics* 15 (3): 675–92. https://doi.org/10.1198/106186006X136976.

Cormack, R. M. 1964. "Estimates of Survival from the Sighting of Marked Animals." *Biometrika* 51 (3/4): 429–38.

Cox, David R. 1972. "Regression Models and Life-Tables." *Journal of the Royal Statistical Society: Series B (Methodological)* 34 (2): 187–202.

Dawid, A Philip. 1982. "The Well-Calibrated Bayesian." *Journal of the American Statistical Association* 77 (379): 605–10.

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.

Efron, Bradley. 1977. "The Efficiency of Cox's Likelihood Function for Censored Data." *Journal of the American Statistical Association* 72 (359): 557–65.

Efron, Bradley, and Robert Tibshirani. 1986. "Bootstrap Methods for Standard Errors, Confidence Intervals, and Other Measures of Statistical Accuracy." *Statistical Science* 1 (1): 54–75.

Efron, Bradley, and Robert J Tibshirani. 1994. *An Introduction to the Bootstrap*. Chapman & Hall/CRC.

Field, Christopher B, Michael J Behrenfeld, James T Randerson, and Paul Falkowski. 1998. "Primary Production of the Biosphere: Integrating Terrestrial and Oceanic Components." *Science* 281 (5374): 237–40.

Fonnesbeck, Chris, Anand Patil, David Huard, and John Salvatier. 2013. *PyMC User's Guide*.

Gabry, Jonah, Daniel Simpson, Aki Vehtari, Michael Betancourt, and Andrew Gelman. 2019. "Visualization in Bayesian Workflow." *Journal of the Royal Statistical Society: Series A (Statistics in Society)* 182 (2): 389–402.

Gelman, A. 2006. "Prior Distributions for Variance Parameters in Hierarchical

Models." *Bayesian Analysis* 1 (3): 515–34.
Gelman, Andrew. 2004. "Parameterization and Bayesian Modeling." *Journal of the American Statistical Association* 99: 537–45.
Gelman, Andrew, J. B. Carlin, Hal S. Stern, David B. Dunson, Aki Vehtari, and Donald B. Rubin. 2013. *Bayesian Data Analysis*. Third Edition. London: Chapman & Hall / CRC Press.
Gelman, Andrew, and Jennifer Hill. 2007. *Data Analysis Using Regression and Multilevel-Hierarchical Models*. Cambridge, United Kingdom: Cambridge University Press.
Gelman, Andrew, Xiao-Li Meng, and Hal Stern. 1996. "Posterior Predictive Assessment of Model Fitness via Realized Discrepancies." *Statistica Sinica*, 733–60.
Gneiting, Tilmann, Fadoua Balabdaoui, and Adrian E Raftery. 2007. "Probabilistic Forecasts, Calibration and Sharpness." *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 69 (2): 243–68.
Gneiting, Tilmann, and Adrian E Raftery. 2007. "Strictly Proper Scoring Rules, Prediction, and Estimation." *Journal of the American Statistical Association* 102 (477): 359–78.
Hairer, Ernst, Syvert P. Nørsett, and Gerhard Wanner. 1993. *Solving Ordinary Differential Equations I: Nonstiff Problems*. 2nd ed. Springer Series in Computational Mathematics, Springer Ser.Comp.Mathem. Hairer,E.:Solving Ordinary Diff. Berlin Heidelberg: Springer-Verlag.
Hindmarsh, Alan C, Peter N Brown, Keith E Grant, Steven L Lee, Radu Serban, Dan E Shumaker, and Carol S Woodward. 2005. "SUNDIALS: Suite of Nonlinear and Differential/Algebraic Equation Solvers." *ACM Transactions on Mathematical Software (TOMS)* 31 (3): 363–96.
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. 2014. "The No-U-Turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo." *Journal of Machine Learning Research* 15: 1593–623. http://jmlr.org/papers/v15/hoffman14a.html.
Huggins, Jonathan H, and Jeffrey W Miller. 2019. "Using Bagged Posteriors for Robust Inference and Model Criticism." *arXiv*, no. 1912.07104.
Jarrett, R. G. 1979. "A Note on the Intervals Between Coal-Mining Disasters." *Biometrika* 66 (1): 191–93.
Jolly, G. M. 1965. "Explicit Estimates from Capture-Recapture Data with Both Death and Immigration-Stochastic Model." *Biometrika* 52 (1/2): 225–47.
Kennedy, Lauren, and Andrew Gelman. 2019. "Know Your Population and Know Your Model: Using Model-Based Regression and Poststratification to Generalize

Findings Beyond the Observed Sample." *arXiv*, no. 1906.11323.
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.
Little, Roderick JA. 1993. "Post-Stratification: A Modeler's Perspective." *Journal of the American Statistical Association* 88 (423): 1001–12.
Lunn, David, Christopher Jackson, Nicky Best, Andrew Thomas, and David Spiegelhalter. 2012. *The BUGS Book: A Practical Introduction to Bayesian Analysis*. CRC Press/Chapman & Hall.
Mazzia, F., J. R. Cash, and K. Soetaert. 2012. "A Test Set for Stiff Initial Value Problem Solvers in the Open Source Software R: Package deTestSet." *Journal of Computational and Applied Mathematics* 236 (16): 4119–31.
Montenbruck, Oliver, and Eberhard Gill. 2000. *Satellite Orbits: Models, Methods and Applications*. Berlin Heidelberg: Springer-Verlag. https://doi.org/10.1007/978-3-642-58351-3.
Muller, Mervin E. 1959. "A Note on a Method for Generating Points Uniformly on n-Dimensional Spheres." *Commun. ACM* 2 (4): 19–20. https://doi.org/10.1145/377939.377946.
Neal, Radford 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.
Park, David K, Andrew Gelman, and Joseph Bafumi. 2004. "Bayesian Multilevel Estimation with Poststratification: State-Level Estimates from National Polls." *Political Analysis* 12 (4): 375–85.
Paustian, Keith, Elissa Levine, Wilfred M Post, and Irene M Ryzhova. 1997. "The Use of Models to Integrate Information and Understanding of Soil C at the Regional Scale." *Geoderma* 79 (1-4): 227–60.

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.

Plackett, Robin L. 1975. "The Analysis of Permutations." *Journal of the Royal Statistical Society Series C: Applied Statistics* 24 (2): 193–202.

Powell, Michael J. D. 1970. "A Hybrid Method for Nonlinear Equations." In *Numerical Methods for Nonlinear Algebraic Equations*, edited by P. Rabinowitz. Gordon; Breach.

Pullin, Jeffrey, Lyle Gurrin, and Damjan Vukcevic. 2021. "Statistical Models of Repeated Categorical Ratings: The r Package Rater." *arXiv* 2010.09335. https://arxiv.org/abs/2010.09335.

Rao, C. Radhakrishna. 1945. "Information and Accuracy Attainable in the Estimation of Statistical Parameters." *Bulletin of the Calcutta Math Society* 37 (3): 81–91.

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.

Riutort-Mayol, Gabriel, Paul-Christian Bürkner, Michael R Andersen, Arno Solin, and Aki Vehtari. 2023. "Practical Hilbert Space Approximate Bayesian Gaussian Processes for Probabilistic Programming." *Statistics and Computing* 33 (1): 17.

Robertson, H. H. 1966. "The Solution of a Set of Reaction Rate Equations." In *Numerical Analysis, an Introduction*, 178–82. Lodon; New York: Academic Press.

Rubin, Donald B. 1984. "Bayesianly Justifiable and Relevant Frequency Calculations for the Applied Statistician." *The Annals of Statistics*, 1151–72.

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 Statistics, University of Otago, Dunedin.

Seber, G. A. F. 1965. "A Note on the Multiple-Recapture Census." *Biometrika* 52 (1/2): 249–59.

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.

Serban, Radu, and Alan C. Hindmarsh. 2021. "Example Programs for IDAS."

LLNL-TR-437091. Lawrence Livermore National Laboratory.
Serban, Radu, Cosmin Petra, Alan C. Hindmarsh, Cody J. Balos, David J. Gardner, Daniel R. Reynolds, and Carol S. Woodward. 2021. "User Documentation for IDAS V5.0.0." Lawrence Livermore National Laboratory.
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.
Talts, Sean, Michael Betancourt, Daniel Simpson, Aki Vehtari, and Andrew Gelman. 2018. "Validating Bayesian Inference Algorithms with Simulation-Based Calibration." *arXiv*, no. 1804.06788.
Timonen, Juho, Nikolas Siccha, Ben Bales, Harri Lähdesmäki, and Aki Vehtari. 2023. "An Importance Sampling Approach for Reliable and Efficient Inference in Bayesian Ordinary Differential Equation Models." *Stat* 12 (1): e614.
Vehtari, Aki, Andrew Gelman, and Jonah Gabry. 2017. "Practical Bayesian Model Evaluation Using Leave-One-Out Cross-Validation and WAIC." *Statistics and Computing* 27 (5): 1413–32.
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.
Zhang, Hao. 2004. "Inconsistent Estimation and Asymptotically Equal Interpolations in Model-Based Geostatistics." *Journal of the American Statistical Association* 99 (465): 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): 7111.