RStan: the R interface to Stan

The Stan Development Team
mc.stanislaw@gmail.com

November 26, 2015

Contents

Abstract

In this vignette we present the RStan package **rstan** for using Stan in R. Stan is a package for making Bayesian inferences using the No-U-Turn sampler (a variant of Hamiltonian Monte Carlo) or frequentist inference via optimization. We illustrate the features of RStan through an example in Gelman et al. (2003).

1 Introduction

Stan is a C++ library for Bayesian modeling and inference that primarily uses the No-U-Turn sampler (NUTS) (Hoffman and Gelman 2012) to obtain posterior simulation given user-specified model and data. Alternatively, Stan can utilize the LBFGS optimization algorithm to maximize an objective function, such as a log-likelihood. The R package, **rstan** allows one to conveniently use Stan from R (R Core Team 2014) and to access Stan output, which includes posterior inferences and also intermediate quantities such as evaluation of the log posterior density and its gradients.

The website for Stan and RStan, `http://mc-stan.org`, provides up-to-date information about how to operate Stan and RStan. For example, "RStan Getting Started" (The Stan Development Team 2014a) has a couple of examples. The present article provides a concise introduction to the functionality of package **rstan** and provides pointers to many functions in **rstan** from the user's perspective.

We start with the prerequisites for using **rstan** (section 1.1) and a typical workflow of using Stan and RStan (section 1.2). In section 2, we illustrate the process of using **rstan** to estimate a Bayesian model. Section 3 presents further details on **rstan**. In section 4, we discuss some functions that **rstan** provides to access the results when Stan is used from the command line.

1.1 Prerequisites

Users need to know how to specify statistical models using the Stan modeling language, which is detailed in the manual of Stan (The Stan Development Team 2014c). We give an example below. To do so, a C++ compiler is required, such

as `g++`[1] or `clang++`[2]. There are instructions on how to install a C++ compiler at `https://github.com/stan-dev/rstan/wiki/RStan-Getting-Started#prerequisites`.

Package **rstan** depends on several other R packages:

- **StanHeaders** which provides the Stan C++ headers
- **BH** which provides Boost C++ headers
- **RcppEigen** which provides Eigen C++ headers
- **Rcpp** which facilitates using C++ from R
- **inline** which compiles C++ for use with R

These package dependencies should be automatically installed if you install the **rstan** package via one of the conventional mechanisms.

1.2 Typical workflow of using RStan

Stan has a modeling language, which is similar to but not identical to that of the Bayesian graphical modeling package BUGS (Lunn et al. 2000). A parser translates the model expressed in the Stan modeling language to C++ code, whereupon it is compiled to an executable program and loaded as a Dynamic Shared Object (DSO) in R and can be called by the user. In summary, the following are typical steps of using Stan for Bayesian inference.

a. Represent a statistical model by writing its log posterior density (up to an arbitrary normalizing constant that does not depend on the unknown parameters in the model) using the Stan modeling language. We recommend a separate text file for this, although it can be done using a character string within R.
b. Translate the model coded in Stan modeling language to C++ code using the `stanc` function (which is called by the `stan` function)
c. Compile the C++ code for the model using a C++ compiler to create a DSO (also called a dynamic link library (DLL)) that can be loaded by R (which is called by the `stan`).
d. Run the DSO to sample from the posterior distribution using the `stan` or `sampling` functions.

[1] `http://gcc.gnu.org`
[2] `http://clang.llvm.org`

e. Diagnose non-convergence of the MCMC chains
f. Conduct inference based on the samples from the posterior distribution

Steps c, d, and e above are all performed implicitly by a single call to `stan`.

2 An example of using rstan

In section 5.5 of Gelman et al. (2003), a hierarchical model is used to model the effect of coaching programs on college admissions tests. The data, shown in Table 1, summarize the results of experiments conducted in eight high schools, with an estimated standard error for each, and these data and model are of historical interest as an example of full Bayesian inference (Rubin 1981). We use this example here for its simplicity and because it represents a nontrivial Markov chain simulation problem in that there is dependence between the parameters of original interest in the study — the effects of coaching in each of the eight schools — and the hyperparameter representing the variation of these effects in the modeled population. Certain implementations of a Gibbs sampler or a Hamiltonian Monte Carlo sampler can be slow to converge in this example. For short, we call this example "eight schools." The statistical model is specified as

$$y_j \sim \text{normal}(\theta_j, \sigma_j), \quad j = 1, \ldots, 8 \tag{1}$$

$$\theta_1, \ldots, \theta_8 \sim \text{normal}(\mu, \tau), \tag{2}$$

in which each σ_j's assumed known and a uniform prior density is used, $p(\mu, \tau) \propto 1$.

2.1 Express the model in Stan

We first need to express this model in the Stan modeling language. The **rstan** package allows a model to be coded in a text file (typically with suffix `.stan`) or in a R character vector (of length one). We put the following text into a file called schools.stan:

```
data {
  int<lower=0> J; // number of schools
  real y[J];      // estimated treatment effects
  real<lower=0> sigma[J]; // s.e. of effect estimates
}
parameters {
```

School	Estimated treatment effect, y_j	Standard error of effect estimate, σ_j
A	28	15
B	8	10
C	-3	16
D	7	11
E	-1	9
F	1	11
G	18	10
H	12	18

Table 1: Observed effects of coaching on college admissions test scores in eight schools. We fit these data using a hierarchical model allowing variation between schools.

```
  real mu;
  real<lower=0> tau;
  vector[J] eta;
}
transformed parameters {
  vector[J] theta;
  theta <- mu + tau * eta;
}
model {
  eta ~ normal(0, 1);
  y ~ normal(theta, sigma);
}
```

The first section of the above code specifies the data that is conditioned upon by Bayes Rule: the number of schools, J; the vector of estimates, $y_1, \ldots, y_J$; and the standard errors, $\sigma_1, \ldots \sigma_J$. Data are labeled as integer or real and can be vectors (or, more generally, arrays) if dimensions are specified. Data can also be constrained; for example, in the above model J has been restricted to be nonnegative and the components of σ_y must all be positive.

The next section of the code defines the parameters whose posterior distribution is sought using Bayes Rule. These are the their mean, μ, and standard deviation, τ, of the school effects, plus the standardized school-level effects η. In this model, we let the undstandardized school-level effects, θ, be a transformed

parameter that uses μ and τ to shift and scale the standardized effects η instead of directly declaring θ as a parameter. By parameterizing the model this way, the sampler runs more efficiently because the resulting multivariate geometry is more amendable to Hamiltonian Monte Carlo (Neal 2011).

Finally, the model block looks similar to standard statistical notation. (Just be careful: the second argument to Stan's normal$(\cdot, \cdot)$ distribution is the standard deviation, not the variance as is usual in statistical notation.) We have written the model in vector notation, which allows Stan to make use of more efficient algorithmic differentiation (AD). It would also be possible — but less efficient — to write the model by replacing `y ~ normal(theta,sigma);` with a loop over the J schools, `for (j in 1:J) y[j] ~ normal(theta[j],sigma[j]);`.

Stan has versions of many of the most useful R functions for statistical modeling, including probability distributions, matrix operations, and special functions. However, the names of the Stan functions may differ from their R counterparts and more subtly, the parameterizations of probability distributions in Stan may differ from those in R for the same distribution. To mitigate this problem, the `lookup` function can be passed a R function or character string naming an R function, and **rstan** will attempt to look up the corresponding Stan function, display its arguments, and give the page number in The Stan Development Team (2014c) where the Stan function is discussed.

```
> lookup("dnorm")

    StanFunction                          Arguments ReturnType Page
344       normal           (reals mu, reals sigma)       real  369
348   normal_log (reals y, reals mu, reals sigma)       real  369
    SamplingStatement
344              TRUE
348             FALSE

> tail(lookup("~")) # looks up all Stan sampling statements

    StanFunction                                            Arguments Page
562    student_t                 (reals nu, reals mu, reals sigma)  372
608      uniform                          (reals alpha, reals beta)  393
618    von_mises                            (reals mu, reals kappa)  391
621      weibull                         (reals alpha, reals sigma)  383
627       wiener (reals alpha, reals tau, reals beta, reals delta)  386
629      wishart                              (real nu, matrix Sigma)  405

> lookup(dwilcox)   # no corresponding Stan function
```

```
[1] "no matching Stan functions"
```

If the `lookup` function fails to find an R function that corresponds to a Stan function, it will treat its argument as a regular expression and attempt to find matches with the names of Stan functions.

2.2 User-defined Stan functions

Stan permits users to define their own functions in a functions block of a Stan program. The functions block is optional but if it exists, it must come before any other block. This mechanism allows users to implement statistical distributions or other functionality that is not currently available in Stan. However, even if the user's function merely wraps calls to existing Stan functions, the code in the model block can be much more readible if several lines of Stan code that accomplish one (or perhaps two) task(s) are replaced by a call to a user-defined function.

Another reason to utilize user-defined functions is that **rstan** provides an `expose_stan_functions` function that exports such functions to the R global environment so that they can be tested in R to ensure that they are working properly. For example,

```
> model_code <-
+ '
+ functions {
+   real standard_normal_rng() {
+     return normal_rng(0,1);
+   }
+ }
+ model {}
+ '
> expose_stan_functions(stanc(model_code = model_code))
> standard_normal_rng(seed = 1)

[1] -0.9529876
```

2.3 Preparing the data

The `stan` function in **rstan** accepts data as a `list` or an `environment`. Alternatively the `data` argument can be omitted and R will search for objects that have the same names as in the `data` block of a Stan program. To prepare the data in R, we create a `list` as follows.

```
> schools_data <-
+   list(J=8,
+   y=c(28,  8, -3,  7, -1,  1, 18, 12),
+   sigma=c(15, 10, 16, 11,  9, 11, 10, 18))
```

It would also be possible (indeed, encouraged) to read in the data from a file rather than to directly enter the numbers in the R script.

2.4 Sample from the posterior distribution

Next, we can call the `stan` function to draw posterior samples:

```
> J <- 8
> y <- c(28,  8, -3,  7, -1,  1, 18, 12)
> sigma <- c(15, 10, 16, 11,  9, 11, 10, 18)
> library(rstan)
> fit1 <- stan(file="schools.stan",
+              # better to add explicitly include: data=schools_data,
+              iter=2000, chains=4, cores=1)
```

Function `stan` wraps the following three steps:

a. Translate a model in Stan code to C++ code
b. Compile the C++ code to a dynamic shared object (DSO) and load the DSO
c. Sample given some user-specified data and other settings

A single call to `stan` performs all three steps, but they can also be executed one by one, which can be useful for debugging. In addition, Stan saves the DSO so that when the same model is fit again (possibly with new data), function `stan` can be called so that only the third step is performed, thus saving compile time.

Function `stan` returns an object of S4[3] class `stanfit`. If no error occurs, the returned `stanfit` object includes the samples drawn from the posterior distribution for the model parameters and other quantities defined in the model. If there is an error (for example, when we have syntax error in our Stan code), `stan` will either quit or return a `stanfit` object that contains no samples. Including the DSO as part of a `stanfit` object allows it to be reused so that compiling the same model could be avoided when we want to sample again with the same or different input of data and other settings. Also if an error happens after the model is

[3]For those who are not familiar with the concept of class and S4 class in R, refer to Chambers (2008). A S4 class consists of some attributes (data) to model an object and some methods to model the behavior of the object. From a user's perspective, once a `stanfit` object is created, we are mainly concerned about what methods are defined for the `stanfit`.

compiled but before sampling (for example, problems with input such as data and initial values), we can reuse the previous compiled model. For class `stanfit`, many methods such as `print` and `plot` are defined to work with the samples and conduct model inference. For example, the following shows a summary of the parameters for our example using function `print`.

```
> print(fit1, pars=c("theta", "mu", "tau", "lp__"),
+       probs=c(.1,.5,.9))

Inference for Stan model: schools.
4 chains, each with iter=2000; warmup=1000; thin=1;
post-warmup draws per chain=1000, total post-warmup draws=4000.

          mean se_mean   sd   10%   50%   90% n_eff Rhat
theta[1] 11.33    0.18 8.26  2.10 10.26 22.10  2192 1.00
theta[2]  7.86    0.10 6.27  0.16  7.86 15.48  4000 1.00
theta[3]  6.28    0.15 7.53 -2.76  6.70 14.70  2510 1.00
theta[4]  7.82    0.11 6.45  0.02  7.68 15.64  3347 1.00
theta[5]  5.06    0.11 6.33 -3.26  5.51 12.53  3118 1.00
theta[6]  6.09    0.12 6.65 -2.04  6.41 14.01  3297 1.00
theta[7] 10.50    0.13 6.83  2.29  9.90 19.45  2727 1.00
theta[8]  8.43    0.15 8.10 -0.76  8.00 18.01  3063 1.00
mu        7.90    0.14 5.05  1.63  7.83 14.26  1343 1.00
tau       6.63    0.17 5.46  1.06  5.37 13.59   976 1.01
lp__     -4.83    0.08 2.67 -8.31 -4.62 -1.61  1134 1.00

Samples were drawn using NUTS(diag_e) at Thu Nov 26 10:26:16 2015.
For each parameter, n_eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor on split chains (at
convergence, Rhat=1).
```

The last line of this output, `lp__`, is the logarithm of the (unnormalized) posterior density as calculated by Stan. This log density can be used in various ways for model evaluation and comparison (see, e.g., Vehtari and Ojanen 2012).

3 Advanced features

In this section, we discuss more details and other advanced features of **rstan**. The details pertain to the optional arguments of the `stan` function, data preprocessing, and methods for the S4 class `stanfit`. In addition, we discuss optimization, which can be used to obtain a point estimates via Stan.

3.1 Arguments to the `stan` function

The primary arguments for sampling (in function `stan` and `sampling`) include data, initial values, and the options of the sampler such as `chains`, `iter`, and `warmup`. In particular, `warmup` specifies the number of iterations that are used by NUTS sampler for the adaptation phase before sampling begins. After the warmup, the sampler turns off adaptation and continues until a total of `iter` iterations have been completed. There is no theoretical guarantee that the samples are drawn from the posterior distribution during warmup, so the warmup samples should only be used for diagnosis and not for inference. The summaries for the parameters shown by the `print` method are calculated using only the samples after warmup.

For function `stan`, argument `init` is used for specifying the initial values. There are several options for `init` and the details can be found in the documentation of the `stan` function. The vast majority of the time, it is adequate to allow Stan to generate its own initial values randomly. However, sometimes it is better to specify the initial values for at least a subset of the objects declared in the `parameters` block of a Stan program.

Stan uses a random number generator (RNG) that supports parallelism. The initialization of the RNG is determined by arguments `seed` and `chain_id`. Even if we are running multiple chains from one call to the `stan`, function we only need to specify one seed, which is randomly generated by R if not specified.

3.2 Data preprocessing and passing

The data passed to `stan` will go through a preprocessing procedure. The details of this preprocessing are documented in the help for function `stan`. Here we stress a few important steps. First, **rstan** allows the user to specify more than what is declared in the data block and anything beyond that is silently omitted. In general, an element in the input R list should be numeric data and its dimension should match the declaration in the data block of the model. So for example, `factor` type in R is not supported as data element for RStan and must be converted to integer codes via `as.integer()`. The Stan modeling language distinguishes between integers and doubles (type `int` and `real` in Stan modeling language, respectively). The `stan` function will convert some R data (which is double-precision usually) to `integers` if possible.

In Stan, we have scalars and other types that are a set of scalars, such as vectors and matrices. As R does not have scalars, **rstan** treats vectors of length

one as scalars. However, we might have a model with data block defined as in Figure 1, in which N can be 1 as a special case. So if we know that N is always larger than 1, we can use a vector of length N in R as the data input for y (for example, a vector created by "`y <- rnorm(N)`"). If we want to prevent **rstan** from treating the input data for y as a scalar when $N = 1$, we need to explicitly make it an array as the following R code shows.

```
> y <- as.array(y)
```

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

Figure 1: Data block of an example model in Stan code

As Stan cannot handle missing values in data automatically, so no element of the data can contain `NA` in R. An important step in **rstan**'s data preprocessing is to check missing values and issue an error if any. To model missing values using Stan, you should create binary indicators of whether a data point is observed or missing and then change the `NA` values in R to valid numbers before calling `stan`.

3.3 Methods for the `stanfit` class

For the fitted object that is an instance of the S4 class `stanfit`, we have defined methods such as `print`, `summary`, `plot`, `pairs`, and `traceplot`. We can use these methods to assess the convergence of the Markov chains by looking at the trace plots and calculating the split $\hat{R}$.[4] The `print` method outputs the mean, standard deviation, quantiles of interest, split $\hat{R}$, and effective sample size for each unknown quantity over all the chains combined.

The `plot` method provides an overview of the output, while the `pairs` method shows two-dimensional density plots for each pair of unknown quantities that are stratified according to the `condition` argument. The `traceplot` method plots the traces of all chains for the specified parameters. If we include the warmup draws by setting `inc_warmup=TRUE` (the default), the background color of the warmup area is different from the post-warmup phase.

[4]Split $\hat{R}$ is an updated version of $\hat{R}$ statistic proposed in Gelman and Rubin (1992) that is based on splitting each chain into two halves. See the Stan manual for more details.

Figure 2 presents the plot of the eight schools example. In this plot, credible intervals (by default 80%) for all the parameters as well as `lp__` (the log of posterior density function up to an additive constant), and the median of each chain are displayed. In addition, under the lines representing intervals, small colored areas are used to indicate which range the value of the split $\hat{R}$ statistic is in. Figure 3 shows the traceplot for the τ parameter.

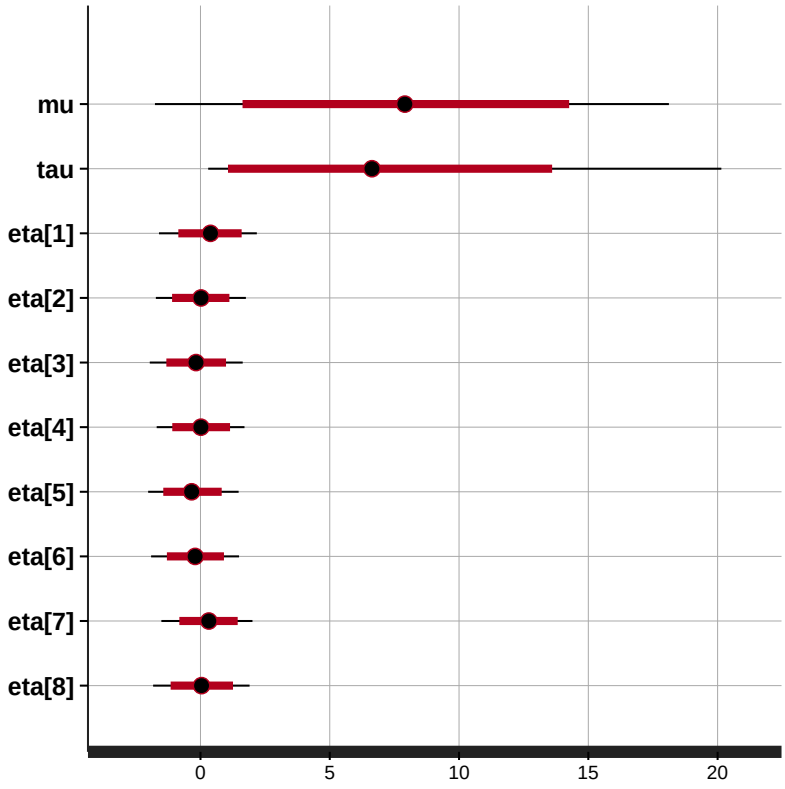

Figure 2: An overview plot for eight schools example

The `stanfit` class has a set of methods to work with the samples drawn from the posterior distribution. First, the `extract` method provides different ways to access the samples. If the argument `permuted` is `TRUE`, then the samples after warmup are returned in an permuted order as a list, each element of which are the samples for a parameter. Here by "one parameter", we mean a scalar/vector/array parameter as a whole defined in the parameters block, transformed parameters block, or generated quantities block of our Stan program. In the eight schools

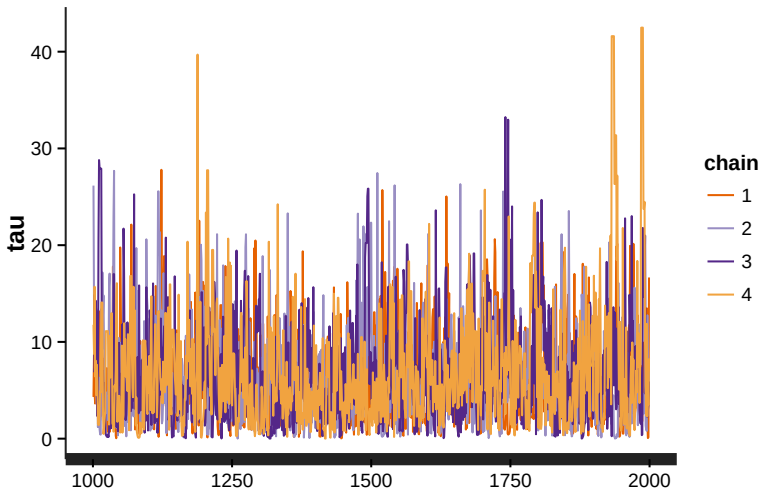

Figure 3: Trace plots of τ in the eight schools model

example, θ is one parameter though it is an array of length J.

If `permuted=FALSE`, the result depends on the `inc_warmup` argument. In either case, the returned object is an array with the first dimension indicating iterations, the second indicating chains, and the third indicating parameters. If `inc_warmup=TRUE`, all iterations are included and if `inc_warmup=FALSE`, only the post-warmup iterations are included. The latter is appropriate for inference, while the former may be useful for diagnosis. In the returned array, each vector/array parameter is "flattened" and are included as the third dimension of the array. In our eight schools examples, the third dimension is `theta[1]`, ..., `theta[8]`.

```
> s <- extract(fit1, pars = c("theta", "mu"), permuted = TRUE)
> names(s)

[1] "theta" "mu"

> dim(s$theta)

[1] 4000    8

> dim(s$mu)

[1] 4000
```

```
> s2 <- extract(fit1, pars = "theta", permuted = FALSE)
> dim(s2)

[1] 1000    4    8

> dimnames(s2)

$iterations
NULL

$chains
[1] "chain:1" "chain:2" "chain:3" "chain:4"

$parameters
[1] "theta[1]" "theta[2]" "theta[3]" "theta[4]" "theta[5]" "theta[6]" "theta[7]"
[8] "theta[8]"
```

In addition, the `as.array`, `as.matrix`, and `as.data.frame` methods are defined for `stanfit` object. These method return the draws of samples in forms of a 3-dimension array, matrix (`rbind`ing the chains), or `data.frame` (that is coerced from a matrix). There are also `dimnames` and `names` methods for `stanfit` objects.

A `stanfit` object keeps all the information regarding the sampling procedure, for example, the model in Stan code, the initial values for all parameters, the seed for the RNG, and parameters used for the sampler (for example, the step size for NUTS) The following methods

1. `get_seed`
2. `get_inits`
3. `get_adaptation_info`
4. `get_sampler_params`

for shown in Table 2 along with other methods defined for the `stanfit` class.

Last, a common feature for many methods that are defined for the `stanfit` class is that the `pars` argument can be specified to indicate a subset of the parameters. This feature is helpful when there are too many parameters in the model or when we need to reduce memory usage. For instance, in the eight schools example, we have parameter θ defined as "`real theta[J]`". So we can specify `pars="theta"` or `pars="theta[1]"`. However, specifying part of θ

(i.e., `pars="theta[1:2]"`) as in R is not allowed — a workaround for this is to specify `pars=c("theta[1]","theta[2]")`. The `stan` function allows the user to specify `pars` so that only part of the samples are returned, which might be problematic from the perspective of diagnosing MCMC convergence since we would apply our diagnostic criterion to a subset of our parameters. To mitigate this loss of diagnostics information, we can use the `get_posterior_mean` function, which returns the posterior mean of all parameters for each chain and all chains combined (excluding warmup samples). Another alternative is to write the samples to external files using the `sample_file` argument of the `stan` function and then conduct diagnostics with the external files.

3.4 Sampling difficulties

The best way to visualize the output of a model is through the **shinyStan** package, which is currently only available from `https://github.com/stan-dev/shinystan`. The **shinyStan** package facilitates both the visualization of parameter distributions and diagnosing problems with a sampler.

However, it is also possible to diagnose problems with a sampler directly via the `get_sampler_params` function.

```
> # all chains combined
> summary(do.call(rbind, args = get_sampler_params(fit1, inc_warmup = TRUE)),
+         digits = 2)

 accept_stat__    stepsize__      treedepth__   n_leapfrog__ n_divergent__
 Min.   :0.00   Min.   : 0.033   Min.   :1.0   Min.   :  1   Min.   :0.000
 1st Qu.:0.81   1st Qu.: 0.301   1st Qu.:3.0   1st Qu.:  7   1st Qu.:0.000
 Median :0.95   Median : 0.376   Median :3.0   Median :  7   Median :0.000
 Mean   :0.84   Mean   : 0.427   Mean   :3.3   Mean   : 10   Mean   :0.011
 3rd Qu.:0.99   3rd Qu.: 0.431   3rd Qu.:4.0   3rd Qu.: 15   3rd Qu.:0.000
 Max.   :1.00   Max.   :13.106   Max.   :8.0   Max.   :223   Max.   :1.000

> # each chain separately
> lapply(get_sampler_params(fit1, inc_warmup = TRUE), summary, digits = 2)

[[1]]
 accept_stat__    stepsize__      treedepth__   n_leapfrog__  n_divergent__
 Min.   :0.00   Min.   : 0.062   Min.   :1.0   Min.   : 1.0   Min.   :0.000
 1st Qu.:0.82   1st Qu.: 0.354   1st Qu.:3.0   1st Qu.: 7.0   1st Qu.:0.000
 Median :0.95   Median : 0.354   Median :3.0   Median : 7.0   Median :0.000
 Mean   :0.84   Mean   : 0.455   Mean   :3.2   Mean   : 9.6   Mean   :0.007
 3rd Qu.:0.99   3rd Qu.: 0.495   3rd Qu.:4.0   3rd Qu.:15.0   3rd Qu.:0.000
```

```
 Max.   :1.00   Max.   :12.220   Max.   :7.0   Max.   :95.0   Max.   :1.000

[[2]]
 accept_stat__    stepsize__     treedepth__   n_leapfrog__ n_divergent__
 Min.   :0.00   Min.   :0.033   Min.   :1.0   Min.   :  1   Min.   :0.000
 1st Qu.:0.87   1st Qu.:0.254   1st Qu.:3.0   1st Qu.:  7   1st Qu.:0.000
 Median :0.97   Median :0.254   Median :4.0   Median : 15   Median :0.000
 Mean   :0.87   Mean   :0.368   Mean   :3.5   Mean   : 12   Mean   :0.009
 3rd Qu.:0.99   3rd Qu.:0.419   3rd Qu.:4.0   3rd Qu.: 15   3rd Qu.:0.000
 Max.   :1.00   Max.   :9.172   Max.   :7.0   Max.   :127   Max.   :1.000

[[3]]
 accept_stat__    stepsize__      treedepth__   n_leapfrog__   n_divergent__
 Min.   :0.00   Min.   : 0.039   Min.   :1.0   Min.   :  1.0   Min.   :0.000
 1st Qu.:0.77   1st Qu.: 0.376   1st Qu.:3.0   1st Qu.:  7.0   1st Qu.:0.000
 Median :0.94   Median : 0.376   Median :3.0   Median :  7.0   Median :0.000
 Mean   :0.82   Mean   : 0.426   Mean   :3.2   Mean   :  9.9   Mean   :0.014
 3rd Qu.:0.99   3rd Qu.: 0.395   3rd Qu.:4.0   3rd Qu.: 15.0   3rd Qu.:0.000
 Max.   :1.00   Max.   :11.245   Max.   :8.0   Max.   :223.0   Max.   :1.000

[[4]]
 accept_stat__    stepsize__      treedepth__   n_leapfrog__  n_divergent__
 Min.   :0.00   Min.   : 0.039   Min.   :1.0   Min.   : 1.0   Min.   :0.000
 1st Qu.:0.78   1st Qu.: 0.420   1st Qu.:3.0   1st Qu.: 7.0   1st Qu.:0.000
 Median :0.94   Median : 0.420   Median :3.0   Median : 7.0   Median :0.000
 Mean   :0.82   Mean   : 0.458   Mean   :3.1   Mean   : 8.6   Mean   :0.013
 3rd Qu.:0.99   3rd Qu.: 0.425   3rd Qu.:3.0   3rd Qu.: 7.0   3rd Qu.:0.000
 Max.   :1.00   Max.   :13.106   Max.   :7.0   Max.   :79.0   Max.   :1.000
```

Here we see that there are a small number of divergent transitions, which are identified by `n_divergent__` being 1. Ideally, there should be no divergent transitions after the warmup phase. The best way to try to eliminate divergent transitions is by increasing the target acceptance probability, which by default is 0.8. Here we see that the mean of `accept_stat__` is close to 0.8 for all chains, but has a very skewed distribution because the median is near 0.95. We could go back and call `stan` again and specify the optional argument `control = list(adapt_delta = 0.9)` to eliminate the divergent transitions. However, sometimes when the target acceptance rate is high, the stepsize is very small and the sampler hits its limit on the number of leapfrog steps it can take per iteration. In this case, it is a non-issue because each chain has a `treedepth__` of at most 7 and the default is 10. But if any `treedepth__` were 11, then it would be wise to increase the limit by passing `control = list(max_treedepth = 12)` (for example) to `stan`.

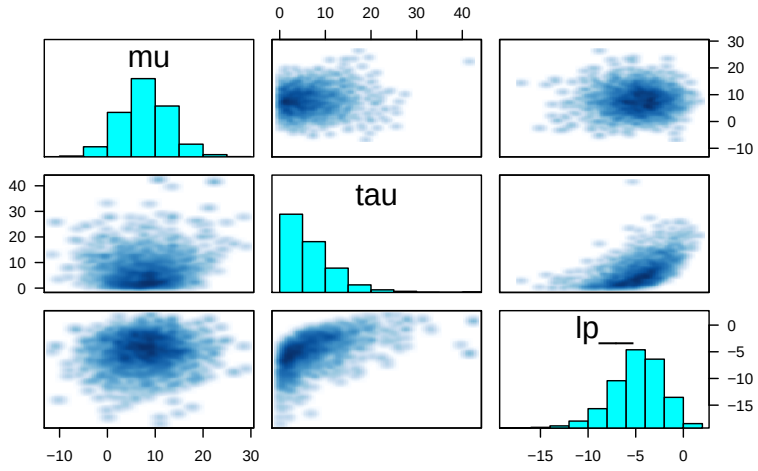

Figure 4: Pairs plots of the common parameters in the eight schools model

Figure 4 gives a graphical representation of the same information. The marginal distribution of each indicated parameter is included as a histogram. By default, draws with below-median `accept_stat__` are plotted below the diagonal and those with above-median `accept_stat__` are plotted above the diagonal. Each off-diagonal square represents a bivariate distribution of the draws for the intersection of the row-variable and the column-variable. Ideally, the below-diagonal intersection and the above-diagonal intersection of the same two variables should have distributions that are mirror images of each other. Any yellow points would indicate transitions where the maximum `treedepth__` was hit, and the red points indicate a transition where `n_divergent__ = 1`. Thus, the pairs plot should be used to get a sense of whether any sampling difficulties are occurring in the tails or near the mode.

3.5 The log posterior function and its gradient

Essentially, we define the log of the probability density function of a posterior distribution up to an unknown additive constant. In Stan, we use `lp__` to represent the realizations of this log kernel at each iteration. In **rstan**, `lp__` is treated as an unknown in the summary and the calculation of split $\hat{R}$ and effective sample size.

A nice feature of **rstan** is that functions for calculating `lp__` and its gradients

for a `stanfit` object are exposed. They are defined for a `stanfit` object, since we need data to create a model instance. These two functions are `log_prob` and `grad_log_prob` respectively. Both take parameters on the *unconstrained* space, even if the support of a parameter is not the whole real line. See The Stan Development Team (2014c) for more details about transformations from the entire real line to some subspace of it. Also the number of unconstrained parameters might be less than the number of parameters. For example, when a parameter is a simplex of length K, the number of unconstrained parameters are $K-1$ due to the constraint that all elements of a simplex must be nonnegative and sum to one. The `get_num_upars` method is provided to get the number of unconstrained parameters, while the `unconstrained_pars` and `constrained_pars` methods can be used to unconstrain or constrain parameters respectively. The former takes a list of parameters as input and transforms it to an unconstrained vector, and the latter does the opposite. Using these functions, we can implement other algorithms such as maximum a posteriori estimation of Bayesian models.

3.6 Optimization in Stan

RStan also provides an interface to Stan's optimizers, which can be used to obtain a point estimate by maximizing the (perhaps penalized) likelihood function defined by a Stan program. We illustrate the feature using a very simple example, estimating the mean from samples assumed to be drawn from normal distribution with known standard deviation. That is, we assume

$$y_1, \ldots, y_n \sim \text{normal}(\mu, 1).$$

By specifying prior of μ with $p(\mu) \propto 1$, the maximum a posteriori estimator for μ is just the sample mean. The following R code shows how to use Stan's optimizers in **rstan**; we first create a `stanmodel` object of **rstan** and then use its `optimizing` method, to which data and other arguments can be fed.

```
> ocode <- "
+   data {
+     int<lower=1> N;
+     real y[N];
+   }
+   parameters {
+     real mu;
+   }
```

Name	Function
`print`	print the summary for parameters obtained using all chains
`summary`	summarize the sample from all chains and individual chains for parameters
`plot`	plot the inferences (intervals, medians, split $\hat{R}$) for parameters
`traceplot`	plot the traces of chains
`pairs`	make a matrix of scatter plots for the samples of parameters
`extract`	extract samples of parameters
`get_stancode`	extract the model code in Stan modeling language
`get_stanmodel`	extract the `stanmodel` object
`get_seed`	get the seed used for sampling
`get_inits`	get the initial values used for sampling
`get_posterior_mean`	get the posterior mean for all parameters
`get_logposterior`	get the log posterior (that is, `lp__`)
`get_sampler_params`	get parameters used by the sampler such as `treedepth` of NUTS
`get_adaptation_info`	get adaptation information of the sampler
`get_num_upars`	get the number of parameters on unconstrained space
`unconstrain_pars`	transform parameter to unconstrained space
`constrain_pars`	transform parameter from unconstrained space to its defined space
`log_prob`	evaluate the log posterior for parameter on unconstrained space
`grad_log_prob`	evaluate the gradient of the log posterior for parameter on unconstrained space

`as.array`
`as.matrix`
`as.data.frame`
	extract the samples excluding warmup to a three dimension array, matrix, data.frame

`dimnames`	obtain the dimension names of the object in its array representation
`names`	obtain the "flattened" parameter names

Table 2: Methods for the S4 class `stanfit`

```
+   model {
+     y ~ normal(mu, 1);
+   }
+ "
> sm <- stan_model(model_code = ocode)
> y2 <- rnorm(20)
> mean(y2)

[1] -0.05048203

> op <- ifelse(grepl('SunOS',Sys.info()['sysname']),
+   "This may not work on Solaris",
+   optimizing(sm, data = list(y = y2, N = length(y2)), hessian = TRUE))

STAN OPTIMIZATION COMMAND (LBFGS)
init = random
save_iterations = 1
init_alpha = 0.001
tol_obj = 1e-12
tol_grad = 1e-08
tol_param = 1e-08
tol_rel_obj = 10000
tol_rel_grad = 1e+07
history_size = 5
seed = 261480211
initial log joint probability = -4.4842
Optimization terminated normally:
  Convergence detected: gradient norm is below tolerance

> print(op)

[[1]]
         mu
-0.05048203
```

3.7 Model compiling in rstan

In RStan, for every model, we use function `stanc` to translate the model from Stan modeling language code to C++ code and then compile the C++ code to dynamic shared object (DSO), which is loaded by R and executed to draw sample. The process of compiling C++ code to DSO, sometimes, takes a while. When the model is the same, we could reuse the DSO from previous run. In function `stan`, if parameter `fit` is specified with a previous fitted object, the compiled model is

reused. When reusing a previous fitted model, we can specify different data and other parameters for function `stan`.

In addition, if fitted models (objects in our working space of R) are saved, for example, by R function `save` and `save.image`, **rstan** is able to save the DSO for models, so that they can be used across R sessions. To (not) save the DSO, specify the `save_dso` argument, which is `TRUE` by default, in the `stan` function.

If the user executes `rstan_options(auto_write = TRUE)`, then a serialized version of the compiled model will be automatically saved to the hard disk in the same directory as the .stan file or in R's temporary directory if the Stan program is expressed as a character string. Although this option is not enabled by default due to CRAN policy, it should ordinarily be specified by users in order to eliminate redundant compilation.

Stan runs much faster when the code is compiled at the maximum level of optimization, which is `-O3` on most C++ compilers. However, the default value is `-O2` in R, which is appropriate for most R packages but entails a slight slowdown for Stan. You can change this default locally by following the instructions at `http://cran.r-project.org/doc/manuals/r-release/R-admin.html#Customizing-package-compilation`. However, you should be advised that setting `CXXFLAGS = -O3` may cause adverse side effects for other R packages.

3.8 Run multiple chains in parallel

For function `stan`, we can specify the number of chains using the `chains` argument. By default, the chains are executed serially (i.e., one at a time) using the parent R process. There is a `cores` argument to `stan` and `sampling` that can be set to the number of chains (if the hardware has sufficient processors and RAM), which is appropriate on most laptops. We ordinarily recommend first calling `options(mc.cores = parallel::detectCores())` once per R session so that `stan` and `sampling` can utilize all available cores.

If you are using another parallelization scheme (perhaps with a remote cluster) **rstan** provides a function called `sflist2stanfit` that consolidates a list of multiple `stanfit` objects (sampled from one model with the same number of warmup and iteration) into one `stanfit` object. It is important to specify the same seed for all the chains and equally important to use a different chain ID (argument `chain_id`). This ensures that the random numbers generated in Stan for all chains are essentially independent. This part is handled automatically by

rstan if `cores > 1`.

4 Working with CmdStan

RStan provides some functions to help use Stan from the command line, CmdStan. First, when Stan reads data or initial values, it supports a subset of the syntax of R dump data formats. So if we use `dump` function in R to prepare data, Stan might not be able to read the data. The `stan_rdump` function in **rstan** dumps the data from R to a format that is supported by Stan with symantics that are very similar to the `dump` function in R.

Second, the `read_stan_csv` function in **rstan** creates a `stanfit` object from reading the comma separated files (CSV) generated by CmdStan. As a result, we can use any methods defined for the `stanfit` class to diagnose and analyze the samples.

5 Summary

In this vignette, we have described the main functionality of RStan from a user's perspective. The help pages with the **rstan** package provide more details for all exposed **rstan** functions. The Stan manual (The Stan Development Team 2014c) provides many details and includes a variety of model examples, many of which are can be executed via function `stan_demo` in the **rstan** package. Finally, the **loo** package, which is on CRAN, is very useful for model comparison using `stanfit` objects.

References

Chambers, J. M. (2008). *Software for Data Analysis : Programming with R*. Springer, New York.

Eddelbuettel, D. and François, R. (2011). Rcpp: Seamless R and C++ integration. *Journal of Statistical Software*, 40(8):1–18.

Gelman, A., Carlin, J. B., Stern, H. S., and Rubin, D. B. (2003). *Bayesian Data Analysis*. CRC Press, London, 2nd edition.

Gelman, A. and Rubin, D. B. (1992). Inference from iterative simulation using multiple sequences. *Statistical Science*, 7(4):457–472.

Hoffman, M. D. and Gelman, A. (2012). The no-U-turn sampler: Adaptively setting path lengths in Hamiltonian Monte Carlo. *Journal of Machine Learning Research*. In press.

Lunn, D., Thomas, A., Best, N., and Spiegelhalter, D. (2000). WinBUGS — a Bayesian modelling framework: Concepts, structure, and extensibility. *Statistics and Computing*, pages 325–337.

Neal, R. (2011). MCMC using Hamiltonian dynamics. In Brooks, S., Gelman, A., Jones, G. L., and Meng, X.-L., editors, *Handbook of Markov Chain Monte Carlo*, pages 116–162. Chapman and Hall/CRC.

Plummer, M. (2011). *JAGS Version 3.1.0 User Manual*.

R Core Team (2014). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0.

Rubin, D. B. (1981). Estimation in parallel randomized experiments. *Journal of educational and behavioral statistics*, 6(4):377–401.

The Stan Development Team (2014a). RStan getting started. https://github.com/stan-dev/rstan/wiki/RStan-Getting-Started.

The Stan Development Team (2014b). Stan: A C++ Library for Probability and Sampling, version 2.6.1. http://mc-stan.org/.

The Stan Development Team (2014c). *Stan Modeling Language: User's Guide and Reference Manual*. Stan Version 2.6.1 (http://mc-stan.org).

Vehtari, A. and Ojanen, J. (2012). A survey of Bayesian predictive methods for model assessment, selection and comparison. *Statistics Surveys*, 6:142–228.

Index