RStan: the R interface to Stan

The Stan Development Team
stan@mc-stan.org

February 6, 2015

Contents

Abstract

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

1 Introduction

Stan is a C++ program for Bayesian modeling and inference that uses the No-U-Turn sampler (NUTS) (Hoffman and Gelman 2012) to obtain posterior simulation given user-specified model and data. The R package, **rstan** allows one to conveniently run Stan within 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, which can be useful in diagnostics. 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) provides information on how to install package **rstan** in R, demonstrating with two examples. The present article provides a complete introduction to the functionality of package **rstan**. As a complement to the documentation about functions and classes in package **rstan**, this vignette 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 and 3, we use an example to illustrate the process of using **rstan** to conduct Bayesian model estimation. Section 4 presents further details on **rstan**. Section 5 describes how to run multiple chains in parallel. In section 6, we discuss some functions that **rstan** provides to help use Stan 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 (**?**). We give an example below.

The package **rstan** cannot currently be installed via CRAN, the "Comprehensive R Archive Network." Instead, they can be installed following the instructions, "Running Stan from R," accessible from `http://mc-stan.org`. As part of

this, a C++ compiler is required (or can be installed following the instruction on that page).

The package **rstan** needs a C++ compiler to compile models the same as in Stan. To install a C++ compiler and make sure that it is accessible in R, refer to "RStan Getting Started" (The Stan Development Team 2014a) and the manual of Stan.

Package **rstan** depends on other two R packages: **Rcpp** (Eddelbuettel and François 2011) and **inline**. **Rcpp** helps bridge the R code and Stan's C++ code. The package **inline** facilitates compiling the C++ code for Stan model and make it possible to run the compiled model in R.

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). Stan uses the program `stanc`[1] to translate a model expressed in Stan modeling language to C++ code. The C++ code is then compiled to an executable program by using a C++ compiler such as `g++`[2] or `clang++`[3]. The resulting program can be executed to draw samples given data and other input. In summary, the following are typical steps of using Stan.

a. Represent a statistical model by writing its log posterior density (up to an arbitrary normalizing constant that does not depends on the unknown parameters in the model); this can be done using Stan modeling language.
b. Translate the model coded in Stan modeling language to C++ code using `stanc`
c. Compile the C++ code for the model using a C++ compiler to create a dynamic shared object (DSO), also called a dynamic link library (DLL), that can be loaded by R.
d. Run the DSO to sample from the posterior distribution
e. Diagnose convergence of the MCMC chains of samples
f. Conduct model inference based on the samples

Steps c, d, and e above are all performed implicitly by a single **rstan** call.

[1] Using Stan from the command line, `stanc` is an executable program.
[2] `http://gcc.gnu.org`
[3] `http://clang.llvm.org`

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.

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 for 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 the Gibbs sampler or Hamiltonian Monte Carlo can be slow to converge in this example. For short, we call this example "eight schools." The statistical model is specified as

$$y_j \sim \text{normal}(\theta_j, \sigma_j^2), \quad j = 1, \ldots, 8 \tag{1}$$

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

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

2.1 Express the model in Stan

We first need to express this model in Stan modeling language. **rstan** allows a model to be coded in a text file (typically with suffix `.stan`) or a character string in R. We put the following plain text in file 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 {
  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 paragraph of the above code specifies the data: 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 code next introduces the parameters: the unknowns to be estimated in the model fit. These are the school effects, θ_j; the mean, μ, and standard deviation, τ, of the population of school effects, the school-level errors η, and the effects, θ. In this model, we let θ be a transformed parameters of μ, τ, and η instead of directly declaring θ as a parameter. By parameterizing this way, the sampler runs more efficiently; the resulting multivariate geometry is better behaved for Hamiltonian Monte Carlo (Neal 2011).

Finally comes the model, which 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 al-

gorithmic differentiation (AD). It would also be possible to write the model more explicitly, for example replacing `y ~ normal(theta,sigma);` with a loop over the J schools, `for (j in 1:J) y[j] ~ normal(theta[j],sigma[j]);`.

2.2 Prepare data

rstan accepts data as a `list` or an `environment`. 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))
```

Often we would already have the elements of the data for a model defined in our workspace. In **rstan**, a convenient way is to provide just the names instead of creating a new list. The following R code demonstrates this feature.

```
> J <- 8
> y <- c(28,  8, -3,  7, -1,  1, 18, 12)
> sigma <- c(15, 10, 16, 11,  9, 11, 10, 18)
> schools_data_nms <- c("J", "y", "sigma")
```

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.

2.3 Sample from the posterior distribution

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

```
> library(rstan)
> fit1 <- stan(file="schools.stan", data=schools_data,
+              iter=100, chains=4)
```

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[4] 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 sample but the DSO. 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 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=100; warmup=50; thin=1;
post-warmup draws per chain=50, total post-warmup draws=200.

          mean se_mean    sd   10%   50%   90% n_eff Rhat
theta[1] 13.48    2.89 10.64  2.54 10.18 27.39    14 1.24
theta[2]  6.83    1.17  6.88 -1.36  6.62 15.24    35 1.08
theta[3]  4.44    0.74  8.36 -4.14  5.04 15.26   127 1.01
theta[4]  7.87    1.09  7.83 -1.18  7.60 16.91    52 1.05
theta[5]  3.77    0.52  6.58 -5.43  4.74 11.02   161 1.00
theta[6]  4.82    0.52  7.07 -2.31  4.92 13.38   184 1.01
theta[7] 11.27    2.16  7.42  3.67  9.44 23.51    12 1.16
theta[8]  8.76    2.91 10.66 -0.59  6.67 20.37    13 1.24
mu       10.86    4.36 10.40  3.81  7.28 24.68     6 2.39
tau       9.84    3.31  9.44  1.18  7.53 25.16     8 1.48
lp__     -4.36    0.55  2.77 -7.90 -3.82 -1.18    25 1.15
```

[4]For those who are not familiar with the concept of class and S4 class in R, refer to Chambers (2008). Simply speaking, a 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 class `stanfit`.

```
Samples were drawn using NUTS(diag_e) at Fri Feb  6 15:58:23 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 while performing the Hamiltonian Monte Carlo algorithm. This log density can be used in various ways for model evaluation and comparison (see, e.g., Vehtari and Ojanen 2012).

3 Eight schools example step by step

In this section, we dive into the functions in **rstan** that implement the three steps of translating, compiling and loading, and sampling. Using these functions individually, we an fit a model in **rstan** in multiple steps as in Stan.

First, we can use `stanc` function to translate the model in Stan modeling language to C++ code. A list is returned from `stanc` with one element being the generated C++ code for the model, which might be helpful for advanced users. When we have syntax errors in the model's Stan code, the error information from `stanc` would be reported to help debug. For the eight schools example, we use

```
> rt <- stanc(file = "schools.stan",
+             model_name = '8schools')
```

Second, after translating a model from Stan code to C++ code, we can use function `stan_model` to compile the C++ code to a DSO and load the DSO into R. In this step (as in calling `stan`), the C++ compiler might spew out a lot of intermediate message such as the C++ code and warning message especially if argument `verbose` is `TRUE`. In most cases, these messages can be ignored unless there is an error.

```
> sm <- stan_model(stanc_ret = rt, verbose = FALSE)
```

Also we can construct a model from the model's Stan code using function `stan_model`. If the input for `stan_model` is given by argument `file` or `model_code` that provides the Stan code for a model, function `stanc` will be called inside `stan_model` to translate the model from Stan code to C++ code.

```
> sm <- stan_model(file="schools.stan",
+                  model_name='schools',
+                  verbose=FALSE)
```

Function `stan_model` returns an object of S4 class `stanmodel` that comprises of mainly the DSO for the model. The most important method defined for S4 class `stanmodel` is `sampling`, which calls Stan's sampler to sample from the posterior distribution with input of data, initial values, and other specification of sampling parameters such as `chains` (number of chains) and `iter` (number of iterations).

```
> fit <- sampling(sm, data=schools_data, chains=4)
```

In fact, the final step in function `stan` is to call `sampling` for a `stanmodel` object created during the process. Method `sampling` returns an object of S4 class `stanfit` as discussed in section 2.3. Also the arguments to control the sampling procedure are the same as those for function `stan`.

4 Advanced features

In this section, we discuss more details and other advanced features of **rstan**. The details regard to the arguments of function `stan`, data preprocessing for the data passed to Stan, and S4 class `stanfit`. Advanced features include using the log posterior function in R defined by a model (with specific data) and the function for computing the gradients. In addition, we discuss optimizer's in Stan, which can be used to obtain a point estimate.

4.1 Arguments of function `stan`

The arguments for sampling (in function `stan` and `sampling`) mainly include data, initial values, and the settings to control the sampler such as `chains`, `iter`, and `warmup`. In particular, `warmup` specifies the number of iterations that are used by NUTS sampler (or other samplers implemented in Stan) in the phase of adaptation. After the warmup, the sampler turns off adaptation. For some sampler, as there is no theoretical guarantee that the samples are drawn from the object distribution during warmup, the samples should not be used for inference. So the summaries for the parameters printed out by methods `print` 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 function `stan`. Here we just point out that currently it does not allow

partially specifying initial values. So if we want to specify the initial values for running a chain, the R list needs to include values for all parameters.

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

Through changing some of augments for calling function `stan`, we can use other samplers implemented by Stan such as HMC (Neal 2011). All the details can be found in the help documentation of function `stan`. As Stan and RStan are being actively developed, other samplers might be added. The help documentation in package **rstan** should always have the up-to-date details.

The test gradient mode in Stan can be used by setting argument `test_grad` to `TRUE`. As in Stan, the test gradient mode will not sample from the posterior distribution, but only print out the gradients for the log probability density function at an initial point calculated by approaches of both the algorithmic differentiation (AD) in Stan and finite difference. The `stanfit` object returned from `stan` will be in test gradient mode and will not contain any sample.

4.2 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 to specify more than what is really needed in that Stan only looks for the data elements from the input R list with the names declared in the data block of the model specification. In general, an element in the input R list should be numeric data and its dimension should match the declaration in the model specification. So for example, `factor` type in R is not supported as data element for RStan. Stan modeling language differentiates data of types of integer and double (type `int` and `real` in Stan modeling language, respectively). But typically in R, we are using type of `double` most of the time since that is the default in R. So an important data preprocessing step inside function `stan` is to convert some data to type of `integer` if possible.

In Stan, we have scalars and other types that are a set of scalars (for example, vectors and matrices). As R does not have scalars, the behavior of **rstan** is to treat vector length 1 to be a scalar. 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, all elements of data cannot contain `NA` in R. An important step in **rstan**'s data preprocessing is to check missing values and issue an error if any.

4.3 Methods of class `stanfit`

For the fitted object represented by S4 class `stanfit`, we have defined methods such as `print`, `summary`, `plot`, and `traceplot`. Using these methods, we first can assess the convergence of the Markov chains by looking at the trace plots and calculating the split $\hat{R}$.[5] The summaries including mean, standard deviation, quantiles of interest, split $\hat{R}$, and effective sample sizes based on the samples after warmup phase can be obtained by `summary` method. Method `print` prints some of the summaries for all chains combined (demonstrated in Section 2) and method `plot` provides an overview plot.

Method `plot` intends to give us an overview of the inference for all the parameters (if possible) in the model. 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 split $\hat{R}$ is in. Method `traceplot` plots the traces of all chains for the parameters specified. If we include the warmup sample by setting `inc_warmup=TRUE` (the default), the background color of the warmup area is different from after warmup. An example for parameter τ in the eight schools example is presented in Figure 3.

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

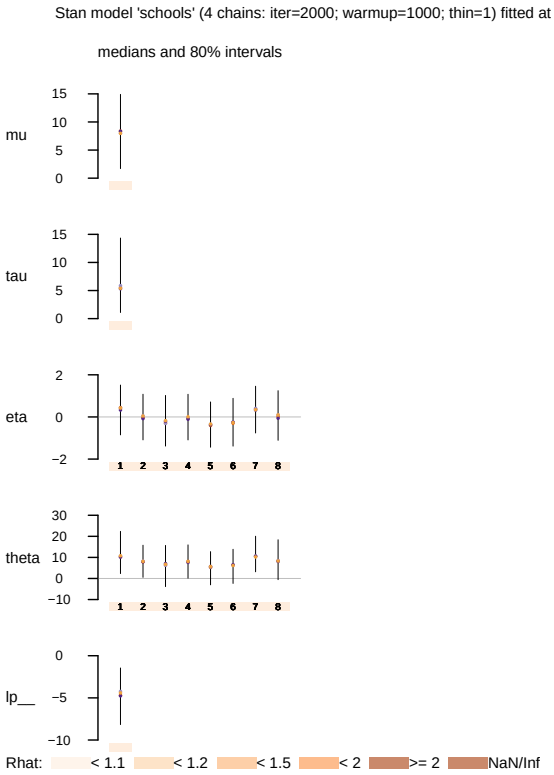

Figure 2: An overview plot for the inference of eight schools example

Class `stanfit` defines a series of methods to work with the samples drawn from the posterior distribution. First, method `extract` provides different ways to access the samples. If argument `permuted` is `TRUE` for calling `extract`, 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 our model. In our eight schools example, θ is one parameter though it is an array of parameters.

When `permuted=FALSE`, we could extract sample for parameters with or without warmup depending on argument `inc_warmup`. In this case, the returned object is an array with the first dimension indicating iterations, the second indicating chains, and the third indicating parameters. Here a vector/array parameter is expanded to their elements. For θ in our eight schools example, they are `theta[1]`,...,`theta[8]`.

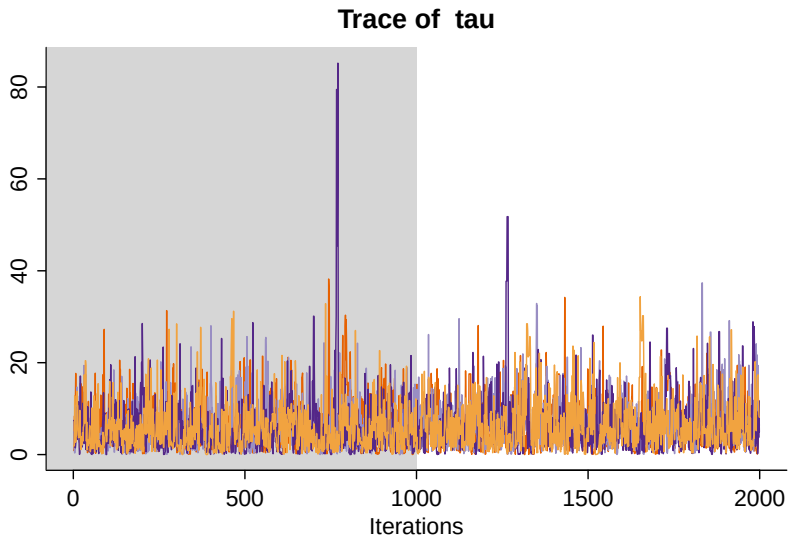

Figure 3: Trace plots of τ in the eight schools model

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

[1] "theta" "mu"

> dim(s$theta)

[1] 4000    8

> dim(s$mu)

[1] 4000

> s2 <- extract(fit, 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]"
[6] "theta[6]" "theta[7]" "theta[8]"
```

In addition, methods `as.array`, `as.matrix`, and `as.data.frame` are defined for `stanfit` object. These method return the draws of samples in forms of a 3-dimension array or a matrix. The three dimensions in order are iterations, chains, and parameters. In the form of a matrix or a data frame, multiple chains are merged. Additionally, method `pairs` creates a matrix of scatter plots of samples, which can help diagnose convergence as well as providing a way to look at the pairwise relationship between parameters.

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) These methods (`get_seed`, `get_inits`, `get_adaptation_info`, and `get_sampler_params`) for obtaining this information are listed in Table 2 along with other methods defined for class `stanfit`.

Last, a common feature for some functions of class `stanfit` is that there is an argument of name `pars`. This argument is used to specify parameters of interest so that a subset of the parameters for a model is, for example, printed (plotted). This feature is helpful when there are too many parameters in the model or when we need to reduce computer 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]")`. Though function `stan` allows to specify `pars` so that only part of the samples are returned, it might be problematic from the perspective of diagnosing MCMC convergence since we would apply our diagnostic criterion to part of our parameters. To mitigate the loss of diagnostics information, we can use function `get_posterior_mean`, which would return the posterior mean of all parameters computed from individual chains and all chains merged excluding warmup samples. A better way for this case is to write samples to external files using argument `sample_file` of function `stan` and then conduct diagnostics using the external files.

4.4 The log posterior function and its gradient

Essentially, we define the log of the probability density of a posterior distribution up to an unknown additive constant. In Stan, we use `lp__` to represent this log density evaluated at each iteration. Often `lp__` becomes one quantity that we are interested in. In **rstan**, `lp__` is treated as if it is a parameter 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 on a `stanfit` object as we need data to create an instance of an abstract model. These two functions are `log_prob` and `grad_log_prob` respectively. Both functions take parameters on the *unconstrained* space, when the support of a parameter is not the whole real line. See **?** for more details about transformation. 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 $K - 1$. Method `get_num_upars` is provided to get the number of unconstrained parameters, To transform parameters between the constrained space and the unconstrained, we can use functions `unconstrained_pars` and `constrained_pars`. The former takes a list of parameters as input and transforms it to unconstrained space, and the latter does the inversion. Using these functions, we can implement other algorithms such as stochastic MAP estimation for Bayesian models.

4.5 Optimization in Stan

RStan also builds the interface to Stan's optimizers, optimization methods built in Stan to obtain a point estimate by maximizing the posterior function defined for a model. 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 MAP estimator for μ would be 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.

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
`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

`pairs`	make a matrix of scatter plots for the samples of parameters

Table 2: Methods of S4 class `stanfit`

```
> ocode <- "
+   data {
+     int<lower=1> N;
+     real y[N];
+   }
+   parameters {
+     real mu;
+   }
+   model {
+     y ~ normal(mu, 1);
+   }
+ "
> sm <- stan_model(model_code = ocode)

TRANSLATING MODEL 'ocode' FROM Stan CODE TO C++ CODE NOW.
COMPILING THE C++ CODE FOR MODEL 'ocode' NOW.

> y2 <- rnorm(20)
> mean(y2)

[1] 0.2417602

> op <- optimizing(sm, data = list(y = y2, N = length(y2)))

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 = 457050644
initial log joint probability = -15.4595
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # ev
       2      -12.3407      0.446773   1.24345e-14           1           1
Optimization terminated normally:
  Convergence detected: gradient norm is below tolerance

> print(op)

$par
       mu
0.2417602

$value
[1] -12.34066
```

4.6 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. Saving the DSO is optional by specifying parameter `save_dso`, which is `TRUE` by default, for function `stan`.

Last, there are some options that configures compiling the C++ code by a compiler such as `g++`. In particular, we can specify the optimization level that is used for a C++ compiler. **rstan** provides function `set_cppo` to configure some of the flags, details of which can be found in the online document. We strongly suggest using `set_cppo('fast')` for better speed of sampling. In addition, we can use `get_cppo` to take a look of current settings and `reset_cppo` to remove the changes by `set_cppo`. Currently, **rstan** implements these functions through modifying `Makevars` file that is typically under folder `.R` in the home directory, so the changes could have side effects to, for example, installing other packages that includes C++ code from source.

5 Run multiple chains in parallel

For function `stan`, we can specify the number of chains using argument `chains`. **rstan** runs chains sequentially (i.e., one at a time) using one R process, which means that **rstan** does not support sampling in parallel directly. But **rstan** provides a function named `sflist2stanfit` to consolidate multiple `stanfit` objects sampled from one model with the same number of warmup and iteration into one `stanfit` object. As a result, if we can run multiple chains in parallel using any approach provided by other packages on one computer (or a cluster), then we can essentially run multiple chains in parallel. For example, we can easily achieve this goal on one computer using package **parallel**, provided within R

base.

Because of the differences in different operating systems (OS), using **parallel** is different for Unix-like operating systems (for example, Linux and Mac OS X) from Microsoft Windows. First, we provide some demonstration code, which uses function `mclapply`, for Unix-like OS.

```
> library(rstan)
> library(parallel)
> f1 <- stan(file="schools.stan",data=schools_data,
+            chains =1, iter=1)
> WINDOWS <- .Platform$OS.type == "windows"
> seed <- 12345
> sflist1 <-
+   mclapply(1:4, mc.cores = ifelse(WINDOWS, 1, 2),
+            function(i) stan(fit = f1, seed = seed,
+                             data = schools_data,
+                             chains = 1, chain_id = i,
+                             refresh = -1))
> fit <- sflist2stanfit(sflist1)
```

For Windows, a different function called `parLapply` in **parallel** can be used since `mclapply` on Windows can use only one core.

```
> library(rstan)
> library(parallel)
> f1 <- stan(file="schools.stan",data=schools_data,
+            chains=1, iter=1)
> seed <- 12345
> num_core = 2
> CL = makeCluster(num_core, outfile = 'cluster.log')
> clusterExport(cl = CL, c("seed", "schools_data", "f1"))
> sflist1 <-
+   parLapply(CL, 1:4,
+             fun = function(i) {
+               require(rstan)
+               stan(fit = f1, seed = seed,
+               data = schools_data,
+               chains = 1, chain_id = i,
+               refresh = -1)
+             })
> fit <- sflist2stanfit(sflist1)
> stopCluster(CL)
```

In the above code, we specify the same seed for all the chains but use different chain ID (argument `chain_id`). This is to make sure that the random numbers

generated in Stan for all chains are independent. Though the above code for Windows is a bit more complicated than the one for Unix, it can also be used for Unix. So the code using `parLapply` is more portable.

6 Work with Stan

RStan provides some functions to help use Stan from the command line. 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 sometimes. Function `stan_rdump` in **rstan** dumps the data in R to the format that is supported by Stan. The usage of this function is very similar to the `dump` function in R.

Second, function `read_stan_csv` in **rstan** creates a `stanfit` object from reading the comma separated files (CSV) generated from using Stan. As a result, we can use many methods defined for class `stanfit` to work with the samples. In other words, we can easily use R to analyze the samples generated by using Stan from the command line.

7 Summary

In this vignetter, we describe the main functionality of RStan from a user perspective. The document within the package should provide more details for all the **rstan** functions. When it comes to Stan, the manual (**?**) provides a lot of details and includes a variety of model examples. **rstan** provides function `stan_demo` to choose and then run an example included in Stan. Some of these examples are detailed in Stan's manual and some of the models originated from the examples of BUGS (Lunn et al. 2000).

As Stan is still being developed, more features will be added. In general, RStan would always implement the interfaces to all the features of Stan and its document would provide all the details. The website of Stan (http://mc-stan.org) provides information on how to get more help.

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 2.6.0. http://mc-stan.org/.

The Stan Development Team (2014c). *Stan Modeling Language: User's Guide and Reference Manual*. Stan Version 2.6.0 (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