# Accessing the contents of a stanfit object

***Stan Development Team***

***2019-07-09***

This vignette demonstrates how to access most of data stored in a stanfit object. A stanfit object (an object of class `"stanfit"`) contains the output derived from fitting a Stan model using Markov chain Monte Carlo or one of Stan's variational approximations (meanfield or full-rank). Throughout the document we'll use the stanfit object obtained from fitting the Eight Schools example model:

```
library(rstan)
fit <- stan_demo("eight_schools", refresh = 0)
```

```
class(fit)
```

```
[1] "stanfit"
attr(,"package")
[1] "rstan"
```

## Posterior draws

There are several functions that can be used to access the draws from the posterior distribution stored in a stanfit object. These are `extract`, `as.matrix`, `as.data.frame`, and `as.array`, each of which returns the draws in a different format.

### extract()

The `extract` function (with its default arguments) returns a list with named components corresponding to the model parameters.

```
list_of_draws <- extract(fit)
```

```
print(names(list_of_draws))
```

```
[1] "mu"    "tau"   "eta"   "theta" "lp__"
```

In this model the parameters `mu` and `tau` are scalars and `theta` is a vector with eight elements. This means that the draws for `mu` and `tau` will be vectors (with length equal to the number of post-warmup iterations times the number of chains) and the draws for `theta` will be a matrix, with each column corresponding to one of the eight components:

```
head(list_of_draws$mu)
```

```
[1] 12.252686 14.907965 -1.373670  5.154424  6.001779 16.319632
```

```
head(list_of_draws$tau)
```

```
[1] 13.510971 10.187284  7.714862 10.243666 16.823645 17.440304
```

```
head(list_of_draws$theta)
```

```
iterations     [,1]      [,2]      [,3]      [,4]      [,5]      [,6]
      [1,] 15.60367  7.588454  1.576459 19.651057  7.664926 1.0059249
      [2,] 15.34390  5.364125 20.853174 14.643021  7.335447 5.0269569
      [3,] 12.34758  8.180855  7.766418  5.044026  8.440089 7.1604064
      [4,] 19.65023 -4.479000 -6.509202 -4.066218 -5.015240 3.7905639
      [5,] 38.23529 -5.416479 -2.989955  6.736584 19.132428 0.8758841
      [6,] 12.03491 11.660757 19.737807 16.787683  7.368761 5.8037854

iterations      [,7]       [,8]
      [1,]  5.284314  15.774425
      [2,] 29.245463  21.365687
      [3,]  8.908072 -11.216071
      [4,]  2.958357   8.495162
      [5,]  8.858071  23.496445
      [6,] 19.364599  25.048120
```

## as.matrix(), as.data.frame(), as.array()

The `as.matrix`, `as.data.frame`, and `as.array` functions can also be used to retrieve the posterior draws from a stanfit object:

```
matrix_of_draws <- as.matrix(fit)
```

```
print(colnames(matrix_of_draws))
```

```
 [1] "mu"       "tau"      "eta[1]"   "eta[2]"   "eta[3]"   "eta[4]"  
 [7] "eta[5]"   "eta[6]"   "eta[7]"   "eta[8]"   "theta[1]" "theta[2]"
[13] "theta[3]" "theta[4]" "theta[5]" "theta[6]" "theta[7]" "theta[8]"
[19] "lp__"    
```

```
df_of_draws <- as.data.frame(fit)
print(colnames(df_of_draws))
```

```
 [1] "mu"       "tau"      "eta[1]"   "eta[2]"   "eta[3]"   "eta[4]"  
 [7] "eta[5]"   "eta[6]"   "eta[7]"   "eta[8]"   "theta[1]" "theta[2]"
[13] "theta[3]" "theta[4]" "theta[5]" "theta[6]" "theta[7]" "theta[8]"
[19] "lp__"    
```

```
array_of_draws <- as.array(fit)
print(dimnames(array_of_draws))
```

```
$iterations
NULL

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

$parameters
 [1] "mu"       "tau"      "eta[1]"   "eta[2]"   "eta[3]"   "eta[4]"  
 [7] "eta[5]"   "eta[6]"   "eta[7]"   "eta[8]"   "theta[1]" "theta[2]"
[13] "theta[3]" "theta[4]" "theta[5]" "theta[6]" "theta[7]" "theta[8]"
[19] "lp__"    
```

The `as.matrix` and `as.data.frame` methods essentially return the same thing except in matrix and data frame form, respectively. The `as.array` method returns the draws from each chain separately and so has an additional dimension:

```
print(dim(matrix_of_draws))
print(dim(df_of_draws))
print(dim(array_of_draws))
```

```
[1] 4000   19
[1] 4000   19
[1] 1000    4   19
```

By default all of the functions for retrieving the posterior draws return the draws for *all* parameters (and generated quantities). The optional argument `pars` (a character vector) can be used if only a subset of the parameters is desired, for example:

```
mu_and_theta1 <- as.matrix(fit, pars = c("mu", "theta[1]"))
head(mu_and_theta1)
```

```
          parameters
iterations        mu   theta[1]
      [1,] 11.594856 14.4775138
      [2,] 14.105900 28.7191144
      [3,]  2.790656  3.5523486
      [4,]  7.954872  0.1407352
      [5,]  6.731532  9.2016480
      [6,]  1.242287 31.3440051
```

## Posterior summary statistics and convergence diagnostics

Summary statistics are obtained using the `summary` function. The object returned is a list with two components:

```
fit_summary <- summary(fit)
print(names(fit_summary))
```

```
[1] "summary"   "c_summary"
```

In `fit_summary$summary` all chains are merged whereas `fit_summary$c_summary` contains summaries for each chain individually. Typically we want the summary for all chains merged, which is what we'll focus on here.

The summary is a matrix with rows corresponding to parameters and columns to the various summary quantities. These include the posterior mean, the posterior standard deviation, and various quantiles computed from the draws. The `probs` argument can be used to specify which quantiles to compute and `pars` can be used to specify a subset of parameters to include in the summary.

For models fit using MCMC, also included in the summary are the Monte Carlo standard error (`se_mean`), the effective sample size (`n_eff`), and the R-hat statistic (`Rhat`).

```
print(fit_summary$summary)
```

```
                mean    se_mean        sd       2.5%        25%
mu        8.40239654 0.23939862 5.6477978 -1.5856605  4.9203079
tau       6.81657128 0.24661168 6.0553821  0.1817417  2.5737606
eta[1]    0.38782960 0.01503165 0.9163859 -1.4184593 -0.2161574
eta[2]   -0.01614640 0.01421635 0.8720442 -1.6918910 -0.6043018
eta[3]   -0.23521562 0.01768225 0.9285214 -1.9991309 -0.8742552
eta[4]   -0.04992150 0.01389894 0.8771066 -1.7692245 -0.6322952
```

```
eta[5]    -0.37046390 0.01595938 0.8976414  -2.0454028  -0.9840520
eta[6]    -0.23426162 0.01543708 0.8877458  -1.9427380  -0.8319003
eta[7]     0.34358924 0.01626435 0.8855897  -1.4118509  -0.2502239
eta[8]     0.03774424 0.01473862 0.9480062  -1.8070263  -0.5896104
theta[1]  11.67308393 0.18109835 8.3804917  -1.6045395   6.1696642
theta[2]   8.07366335 0.09534171 6.2302621  -3.8610079   4.0153215
theta[3]   5.99503306 0.14031161 7.8550471 -12.0653111   1.8476810
theta[4]   7.82824007 0.11181224 6.5417373  -5.0297220   3.7450550
theta[5]   5.09580003 0.10321987 6.5831540  -9.4541310   1.2067127
theta[6]   6.28507381 0.11712779 6.8298216  -8.4880139   2.3255196
theta[7]  10.86160511 0.15631508 6.8499727  -0.9571675   6.2709256
theta[8]   8.58388560 0.19459084 8.0906663  -6.6119231   3.6710090
lp__     -39.55676603 0.07746429 2.7015009 -45.5703000 -41.1940633
                  50%        75%      97.5%     n_eff      Rhat
mu         8.20833569 11.5279047  20.510302  556.5638 1.0051426
tau        5.35894999  9.3390741  22.420817  602.9146 1.0089599
eta[1]     0.39503008  1.0036727   2.199559 3716.5797 1.0001790
eta[2]    -0.03301239  0.5607387   1.736386 3762.7118 0.9998571
eta[3]    -0.25034213  0.3901198   1.616674 2757.4571 1.0015253
eta[4]    -0.05807728  0.5077293   1.711333 3982.3682 1.0016549
eta[5]    -0.39406939  0.2151956   1.474333 3163.5424 0.9994128
eta[6]    -0.23986139  0.3256300   1.576667 3307.0963 1.0002083
eta[7]     0.33572257  0.9322493   2.075483 2964.7765 0.9998975
eta[8]     0.03800369  0.6691663   1.912977 4137.2222 0.9995512
theta[1]  10.52117034 15.6999619  32.100103 2141.4602 1.0024168
theta[2]   8.05066651 11.9398324  21.131261 4270.1858 0.9993536
theta[3]   6.59203659 10.8666143  19.738102 3134.0820 0.9996507
theta[4]   7.84118702 11.7344730  21.401794 3423.0049 0.9999652
theta[5]   5.57323870  9.4067905  16.804645 4067.6299 0.9997760
theta[6]   6.43362090 10.5563822  19.225530 3400.1558 1.0011304
theta[7]  10.31091633 14.6990088  26.365838 1920.3302 1.0029145
theta[8]   8.24268197 12.8361120  25.946117 1728.7166 1.0018890
lp__     -39.24268861 -37.6783486 -34.997607 1216.2059 1.0008591
```

If, for example, we wanted the only quantiles included to be 10% and 90%, and for only the parameters included to be `mu` and `tau`, we would specify that like this:

```
mu_tau_summary <- summary(fit, pars = c("mu", "tau"), probs = c(0.1, 0.9))$summary
print(mu_tau_summary)
```

```
        mean   se_mean       sd       10%      90%    n_eff     Rhat
mu  8.402397 0.2393986 5.647798 1.8414735 14.75461 556.5638 1.005143
tau 6.816571 0.2466117 6.055382 0.9959847 14.28546 602.9146 1.008960
```

Since `mu_tau_summary` is a matrix we can pull out columns using their names:

```
mu_tau_80pct <- mu_tau_summary[, c("10%", "90%")]
print(mu_tau_80pct)
```

```
          10%      90%
mu  1.8414735 14.75461
tau 0.9959847 14.28546
```

## Sampler diagnostics

For models fit using MCMC the stanfit object will also contain the values of parameters used for the sampler. The `get_sampler_params` function can be used to access this information.

The object returned by `get_sampler_params` is a list with one component (a matrix) per chain. Each of the matrices has number of columns corresponding to the number of sampler parameters and the column names provide the parameter names. The optional argument inc_warmup (defaulting to `TRUE`) indicates whether to include the warmup period.

```
sampler_params <- get_sampler_params(fit, inc_warmup = FALSE)
sampler_params_chain1 <- sampler_params[[1]]
colnames(sampler_params_chain1)
```

```
[1] "accept_stat__" "stepsize__"    "treedepth__"   "n_leapfrog__"
[5] "divergent__"   "energy__"
```

To do things like calculate the average value of `accept_stat__` for each chain (or the maximum value of `treedepth__` for each chain if using the NUTS algorithm, etc.) the `sapply` function is useful as it will apply the same function to each component of `sampler_params`:

```
mean_accept_stat_by_chain <- sapply(sampler_params, function(x) mean(x[,
"accept_stat__"]))
print(mean_accept_stat_by_chain)
```

```
[1] 0.7945660 0.8666220 0.9066320 0.8979723
```

```
max_treedepth_by_chain <- sapply(sampler_params, function(x) max(x[, "treedepth__"]))
print(max_treedepth_by_chain)
```

```
[1] 5 5 5 5
```

## Model code

The Stan program itself is also stored in the stanfit object and can be accessed using `get_stancode`:

```
code <- get_stancode(fit)
```

The object `code` is a single string and is not very intelligible when printed:

```
print(code)
```

```
[1] "data {\n  int<lower=0> J;          // number of schools \n  real y[J];
// estimated treatment effects\n  real<lower=0> sigma[J];  // s.e. of effect estimates
\n}\nparameters {\n  real mu; \n  real<lower=0> tau;\n  vector[J] eta;\n}\ntransformed
parameters {\n  vector[J] theta;\n  theta = mu + tau * eta;\n}\nmodel {\n  target +=
normal_lpdf(eta | 0, 1);\n  target += normal_lpdf(y | theta, sigma);\n}"
attr(,"model_name2")
[1] "schools"
```

A readable version can be printed using `cat`:

```
cat(code)
```

```
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 {
  target += normal_lpdf(eta | 0, 1);
  target += normal_lpdf(y | theta, sigma);
}
```

## Initial values

The `get_inits` function returns initial values as a list with one component per chain. Each component is itself a (named) list containing the initial values for each parameter for the corresponding chain:

```
inits <- get_inits(fit)
inits_chain1 <- inits[[1]]
```

```
print(inits_chain1)
```

```
$mu
[1] -0.201507

$tau
[1] 0.1682838

$eta
[1] -1.6057067 -1.8569811  0.3266007  1.0876453 -0.7585606 -0.4009873
[7] -1.7051884 -0.8772185

$theta
[1] -0.47172139 -0.51400680 -0.14654537 -0.01847388 -0.32916042 -0.26898663
[7] -0.48846255 -0.34912864
```

## (P)RNG seed

The `get_seed` function returns the (P)RNG seed as an integer:

```
print(get_seed(fit))
```

```
[1] 886183407
```

## Warmup and sampling times

The `get_elapsed_time` function returns a matrix with the warmup and sampling times for each chain:

```
print(get_elapsed_time(fit))
```

```
          warmup   sample
chain:1 0.059150 0.034474
chain:2 0.036554 0.035265
chain:3 0.039221 0.040196
chain:4 0.036331 0.039571
```