1.5 Practical: first Stan model

Benjamin Rosenbaum

October 24, 2022

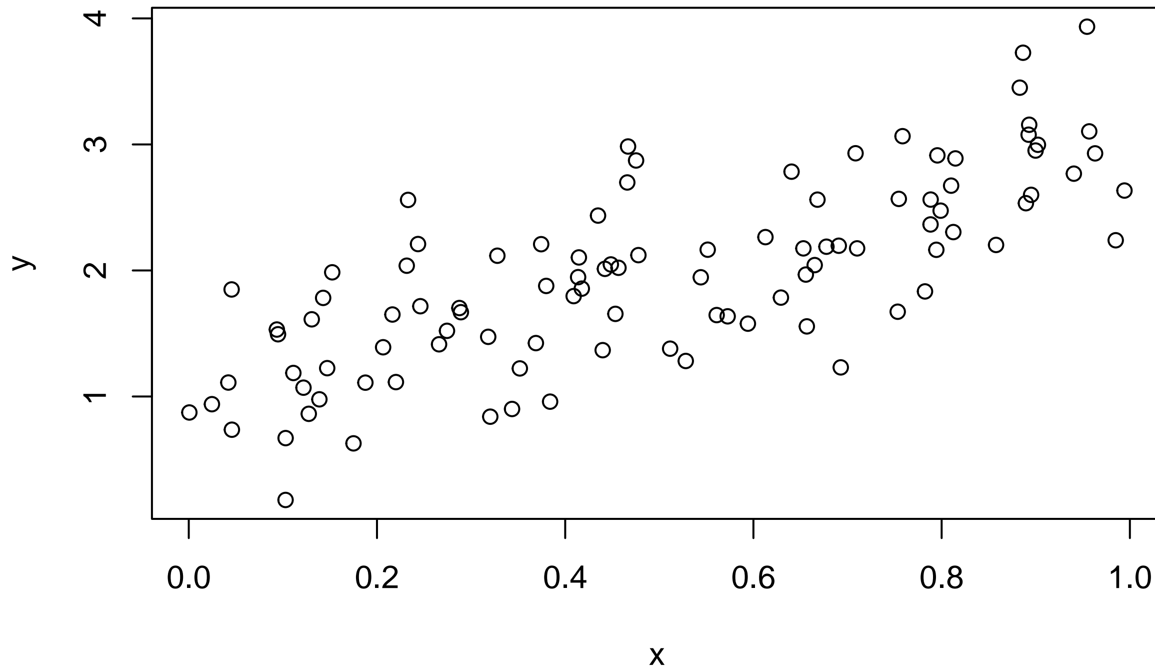
Setup

```
rm(list=ls())
library(rstan)
library(coda)
set.seed(123) # initiate random number generator for reproducibility
```

Generate data

We simulate sample data for linear regression

```
n=100

a=1
b=2
sigma=0.5

x = runif(n=n, min=-0, max=1)
y = rnorm(n=n, mean=a+b*x, sd=sigma)

df = data.frame(x=x,
                y=y)

plot(df)
```



Statistical model, deterministic and stochastic part

Statistical model for linear regression

$$y_i \sim \text{normal}(\mu_i, \sigma)$$
$$\mu_i = a + b \cdot x_i$$

Stan model

The Stan code can be written in the R script and is saved as text using `'...'`.

We use the 3 blocks `data{}`, `parameters{}` and `model{}`.

The first 2 blocks just contain the definition of all variables.

Note that σ is constrained to be positive.

In the model block we define priors for all parameters and the likelihood function, which contains a deterministic part `a+b*x` and a stochastic part `y~normal(..., sigma)`

We can use `//` to comment the code (same as `#` in R).

The likelihood definition `y ~ normal(a+b*x,sigma);` is short for `for(i in 1:n){ y[i]~normal(a+b*x[i],sigma); }`

We use (very) weakly informative priors (normal distribution with zero mean and sd of 10).

Notice that sigma has a lower boundary of 0, i.e. the prior for sigma is a positive half-normal distribution.

We use a general expectation as prior information (e.g. intercept of 100 wouldn't make sense), but not summary statistics of the data (e.g. slope from `lm()`-fit etc. - please never do that!)

More on priors tomorrow!

```
stan_code = '
data {
  int n;
```

```
  vector[n] x;
  vector[n] y;
}
parameters {
  real a; // intercept
  real b; // slope
  real<lower=0> sigma;  // standard deviation
}
model {
  // priors
  a ~ normal(0, 10);
  b ~ normal(0, 10);
  sigma ~ normal(0, 10);
  // likelihood
  y ~ normal(a+b*x, sigma);
}
'
```

Prepare data

The data have to be prepared in R according to the `data{}` block in the Stan code. It is saved as a named list.

```
data = list(n=n,
            x=df$x,
            y=df$y)
data
```

```
## $n
## [1] 100
##
## $x
##   [1] 0.2875775201 0.7883051354 0.4089769218 0.8830174040 0.9404672843
##   [6] 0.0455564994 0.5281054880 0.8924190444 0.5514350145 0.4566147353
##  [11] 0.9568333453 0.4533341562 0.6775706355 0.5726334020 0.1029246827
##  [16] 0.8998249704 0.2460877344 0.0420595335 0.3279207193 0.9545036491
##  [21] 0.8895393161 0.6928034062 0.6405068138 0.9942697766 0.6557057991
##  [26] 0.7085304682 0.5440660247 0.5941420204 0.2891597373 0.1471136473
##  [31] 0.9630242325 0.9022990451 0.6907052784 0.7954674177 0.0246136845
##  [36] 0.4777959711 0.7584595375 0.2164079358 0.3181810076 0.2316257854
##  [41] 0.1428000224 0.4145463358 0.4137243263 0.3688454509 0.1524447477
##  [46] 0.1388060634 0.2330340995 0.4659624503 0.2659726404 0.8578277153
##  [51] 0.0458311667 0.4422000742 0.7989248456 0.1218992600 0.5609479838
##  [56] 0.2065313896 0.1275316502 0.7533078643 0.8950453592 0.3744627759
##  [61] 0.6651151946 0.0948406609 0.3839696378 0.2743836446 0.8146400389
##  [66] 0.4485163414 0.8100643530 0.8123895095 0.7943423211 0.4398316876
##  [71] 0.7544751586 0.6292211316 0.7101824014 0.0006247733 0.4753165741
##  [76] 0.2201188852 0.3798165377 0.6127710033 0.3517979092 0.1111354243
##  [81] 0.2436194727 0.6680555874 0.4176467797 0.7881958340 0.1028646443
##  [86] 0.4348927415 0.9849569800 0.8930511144 0.8864690608 0.1750526503
##  [91] 0.1306956916 0.6531019250 0.3435164723 0.6567581280 0.3203732425
##  [96] 0.1876911193 0.7822943013 0.0935949867 0.4667790416 0.5115054599
##
## $y
```

```
##   [1] 1.7018143 2.5623369 1.7965186 3.4503360 2.7680491 1.8493483 1.2818346
##   [8] 3.0771450 2.1647972 2.0212003 3.1034864 1.6555066 2.1885376 1.6359791
##  [15] 0.6699538 2.9514143 1.7162804 1.1106212 2.1169752 3.9340496 2.5335630
##  [22] 1.2310224 2.7838829 2.6339392 1.9674073 2.9298466 1.9457455 1.5779252
##  [29] 1.6689712 1.2247816 2.9289306 2.9972383 2.1960805 2.9131231 0.9389841
##  [36] 2.1214829 3.0653386 1.6504066 1.4733962 2.0376554 1.7823520 2.1032912
##  [43] 1.9468145 1.4237379 1.9852157 0.9774823 2.5597347 2.6982302 1.4140951
##  [50] 2.2024450 0.7364591 2.0128420 2.4745038 1.0700272 1.6460867 1.3905489
##  [57] 0.8626111 1.6726448 2.5999775 2.2084239 2.0425569 1.4936635 0.9589979
##  [64] 1.5209863 2.8889837 2.0476094 2.6729668 2.3044260 2.1638325 1.3675990
##  [71] 2.5677736 1.7847050 2.1750861 0.8732035 2.8725642 1.1142628 1.8773264
##  [78] 2.2645224 1.2226675 1.1866168 2.2095144 2.5618632 1.8559100 2.3651433
##  [85] 0.1791057 2.4354541 2.2395939 3.1560760 3.7274899 0.6281587 1.6122836
##  [92] 2.1751051 0.9009609 1.5561824 0.8399784 1.1099290 1.8337108 1.5311484
##  [99] 2.9836126 1.3794957
```

Compile model (translates to fast C++ code)

Usually, we run more than 1 MCMC chain. These calculations are totally independent from each other and can be run simultaneously. `detectCores()` shows the number of available CPU cores.

```
parallel::detectCores(all.tests = FALSE, logical = TRUE)
```

```
## [1] 8
```

Now we tell Stan how many CPU cores we want to use. We want to run 3 MCMC chains in parallel, so we set this number to 3.

```
rstan_options(auto_write = TRUE)
options(mc.cores = 3) # number of CPU cores used
```

Until yet, the Stan code is just saved as text. `stan_model()` compiles this text into fast C++ code and we save this as an object. This can take a minute!

```
stan_model = stan_model(model_code=stan_code)
```

```
## Trying to compile a simple C file
```

```
## Running /usr/lib/R/bin/R CMD SHLIB foo.c
## gcc -I"/usr/share/R/include" -DNDEBUG   -I"/usr/lib/R/site-library/Rcpp/include/"  -I"/usr/lib/R/site
## In file included from /usr/lib/R/site-library/RcppEigen/include/Eigen/Core:88,
##                  from /usr/lib/R/site-library/RcppEigen/include/Eigen/Dense:1,
##                  from /usr/lib/R/site-library/StanHeaders/include/stan/math/prim/mat/fun/Eigen.hpp:13
##                  from <command-line>:
## /usr/lib/R/site-library/RcppEigen/include/Eigen/src/Core/util/Macros.h:628:1: error: unknown type nam
##   628 | namespace Eigen {
##       | ^~~~~~~~~
## /usr/lib/R/site-library/RcppEigen/include/Eigen/src/Core/util/Macros.h:628:17: error: expected ‘=’,
##   628 | namespace Eigen {
##       |                 ^
## In file included from /usr/lib/R/site-library/RcppEigen/include/Eigen/Dense:1,
##                  from /usr/lib/R/site-library/StanHeaders/include/stan/math/prim/mat/fun/Eigen.hpp:13
##                  from <command-line>:
## /usr/lib/R/site-library/RcppEigen/include/Eigen/Core:96:10: fatal error: complex: No such file or di
##    96 | #include <complex>
##       |          ^~~~~~~~~
## compilation terminated.
```

```
## make: *** [/usr/lib/R/etc/Makeconf:168: foo.o] Error 1
```

MCMC sampling

Now we can start the MCMC sampler using `sampling()`.

We hand over the compiled object and the data.

Additionally, we tell the sampler how many Markov chains to run, how many iterations per chain, and how many of these are just used for warmup (discarded afterwards).

The total number of samples will be `chains*(iter-warmup)`.

More options see help function `?sampling`.

One important option we didn't use is specifying initial values for the chains.

If it is unspecified, Stan uses random initial values from the interval $[-2, 2]$.

```
fit  = sampling(stan_model,
                data=data,
                chains=3,
                iter=2000,
                warmup=1000
)
```

Printing the results gives is information on the marginal (i.e. 1d / single parameter) estimates (remember, the posterior distribution is multidimensional):

mean, standard deviation and quantiles / confidence intervals.

Convergence of the MCMC sampling is monitored by `n_eff` and `Rhat`.

`n_eff` is the number of effective (uncorrelated) samples of the posterior and is generally lower than `n_total`.

But if `n_eff` was only a small fraction of `n_total`, i.e. 1%, this would indicate some problems with our model.

`Rhat` checks if all chains behave similarly, values close to one (<1.1) indicate good behavior.

The last "parameter" is the log posterior denstity, we will ignore it for now.

```
print(fit)
```

```
## Inference for Stan model: 8682ad33e7880da14b21e5436e1aa3bf.
## 3 chains, each with iter=2000; warmup=1000; thin=1;
## post-warmup draws per chain=1000, total post-warmup draws=3000.
##
##        mean se_mean   sd  2.5%   25%   50%   75% 97.5% n_eff Rhat
## a      0.99    0.00 0.10  0.79  0.92  0.99  1.06  1.19  1299    1
## b      1.96    0.00 0.17  1.63  1.84  1.96  2.08  2.30  1282    1
## sigma  0.49    0.00 0.04  0.43  0.47  0.49  0.51  0.56  1255    1
## lp__  21.15    0.04 1.23 17.87 20.58 21.46 22.05 22.58  1128    1
##
## Samples were drawn using NUTS(diag_e) at Mon Oct 17 11:19:48 2022.
## 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).
```

You can also specify the output, printing only some parameters or only 95% confidence intervals.

Btw, the choice of 95% confidence interval is rather arbitrary.

We can also print a 90% confidence interval (5% cut off at both sides).

```
print(fit,
      digits=3,
      probs=c(0.05, 0.95),
      pars=c("a","b","sigma")
      )
```

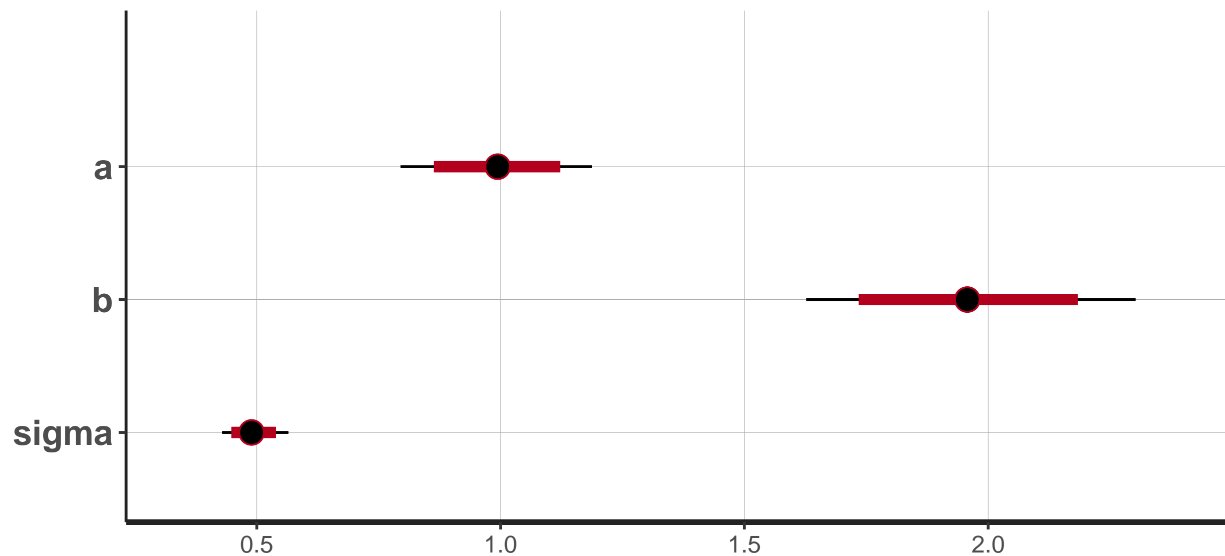
```
## Inference for Stan model: 8682ad33e7880da14b21e5436e1aa3bf.
## 3 chains, each with iter=2000; warmup=1000; thin=1;
## post-warmup draws per chain=1000, total post-warmup draws=3000.
##
##        mean se_mean    sd    5%   95% n_eff  Rhat
## a     0.993   0.003 0.100 0.826 1.156  1299 1.001
## b     1.960   0.005 0.174 1.676 2.246  1282 1.002
## sigma 0.492   0.001 0.036 0.436 0.555  1255 1.004
##
## Samples were drawn using NUTS(diag_e) at Mon Oct 17 11:19:48 2022.
## 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 posterior distribution can also be plotted.

```
plot(fit)
```

```
## ci_level: 0.8 (80% intervals)
```

```
## outer_level: 0.95 (95% intervals)
```

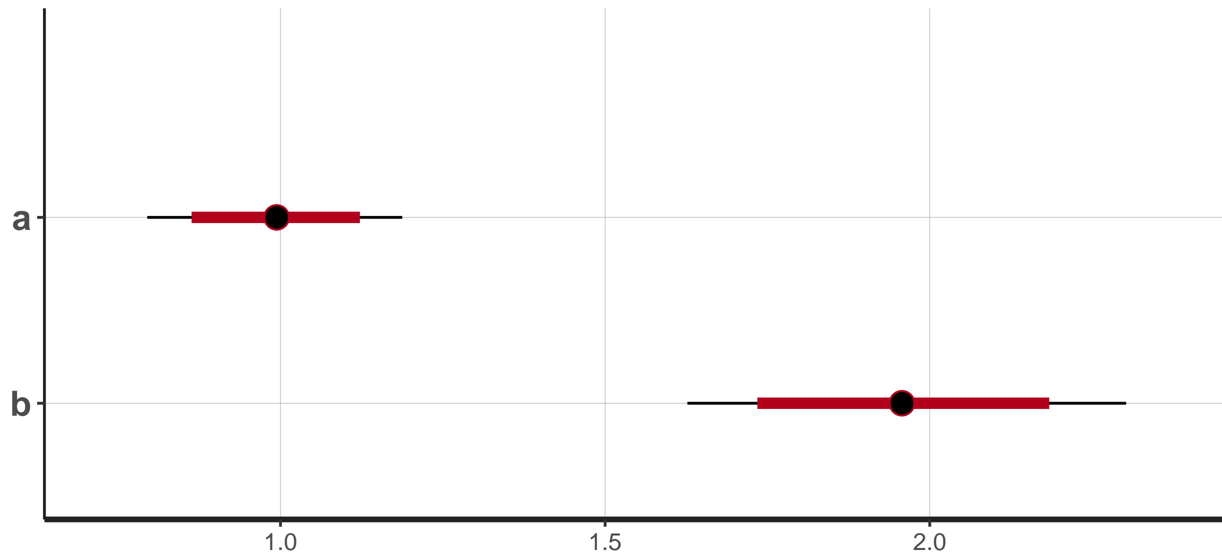


Or just for certain parameters:

```
plot(fit, pars=c("a","b"))
```

```
## ci_level: 0.8 (80% intervals)
```

```
## outer_level: 0.95 (95% intervals)
```



There is another option from the coda package. First we transform the stan object

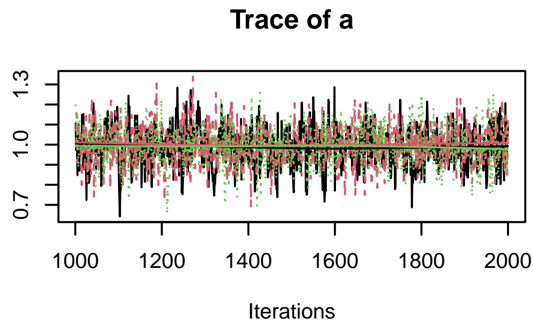
```
posterior = As.mcmc.list(fit)
```

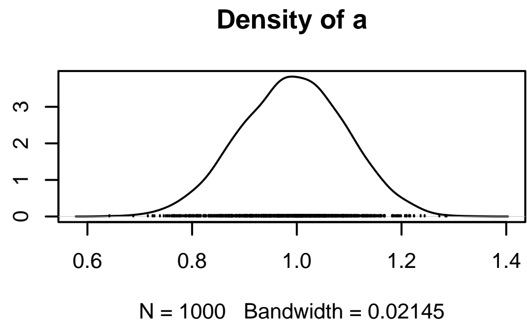
Plotting this object shows us traceplots of the chains and marginal density functions of the posterior distribution.

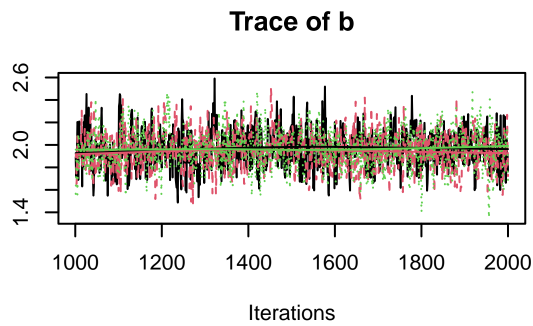
The chains are plotted in different colours. We also can visually inspect if all chains behave similarly.

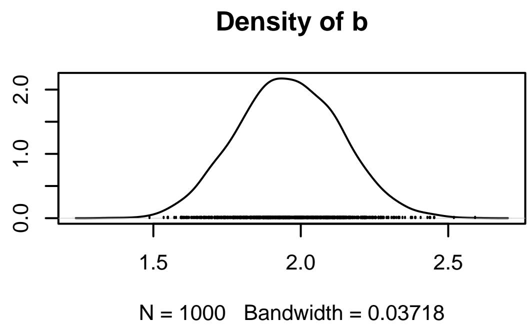
They should look like a "fat hairy caterpillar".

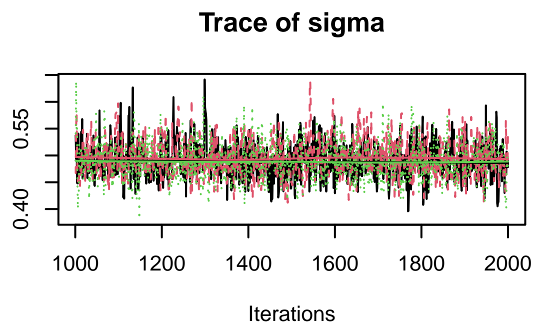
```
plot(posterior)
```

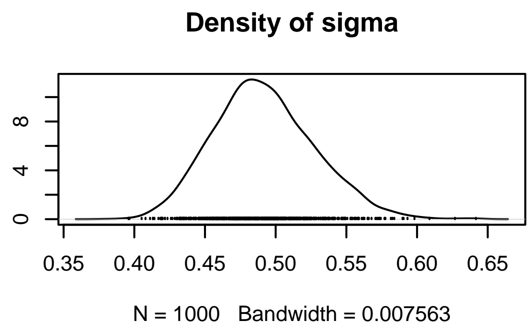


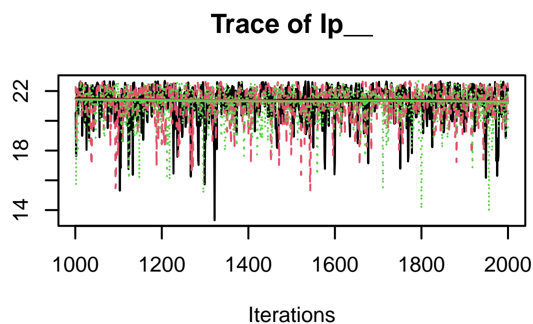


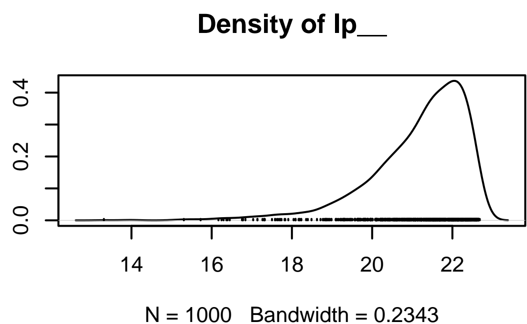








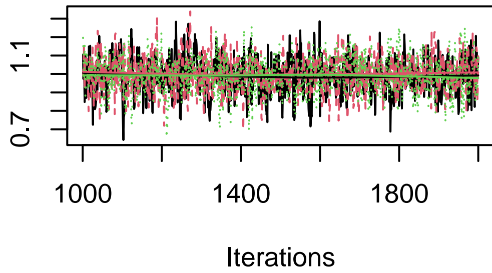




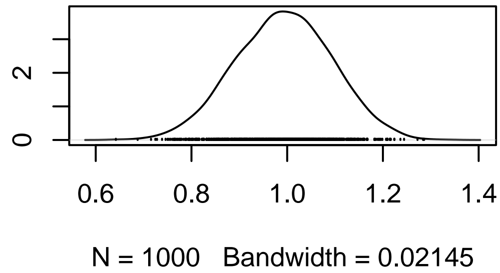
You can also specify which parameters to plot, either indicing by name or number.

```
plot(posterior[, c("a", "b")])
plot(posterior[, 1:2])
```

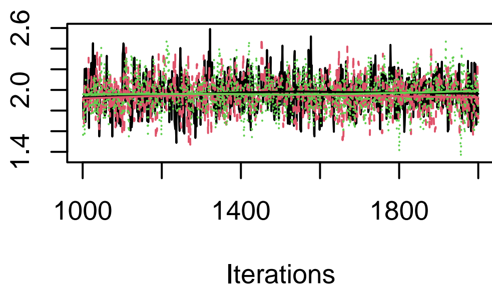
Trace of a



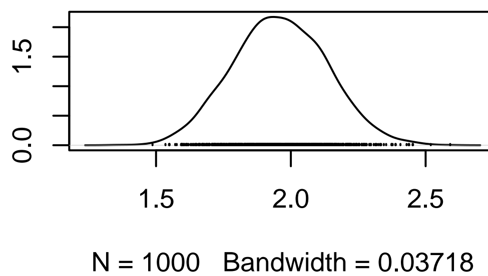
Density of a



Trace of b

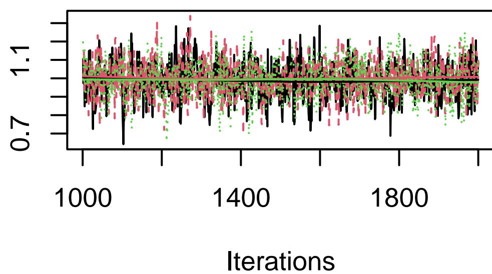


Density of b

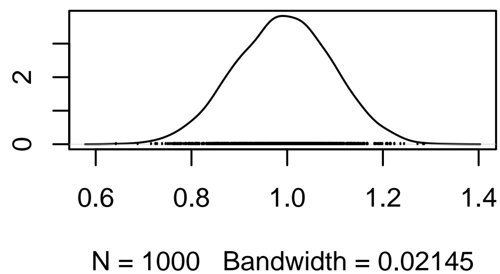



```
plot(posterior[, 1:2])
```

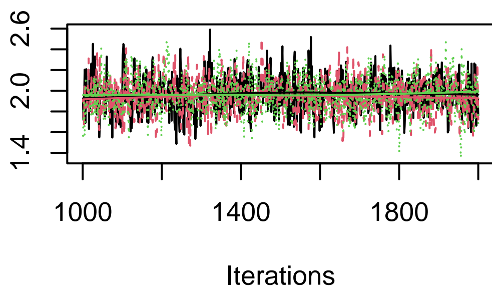
Trace of a



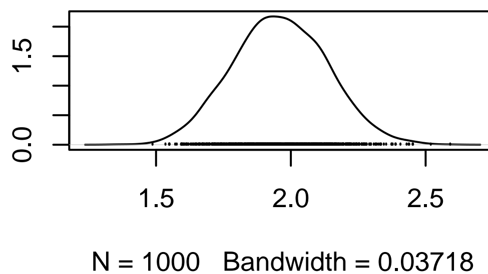
Density of a



Trace of b



Density of b

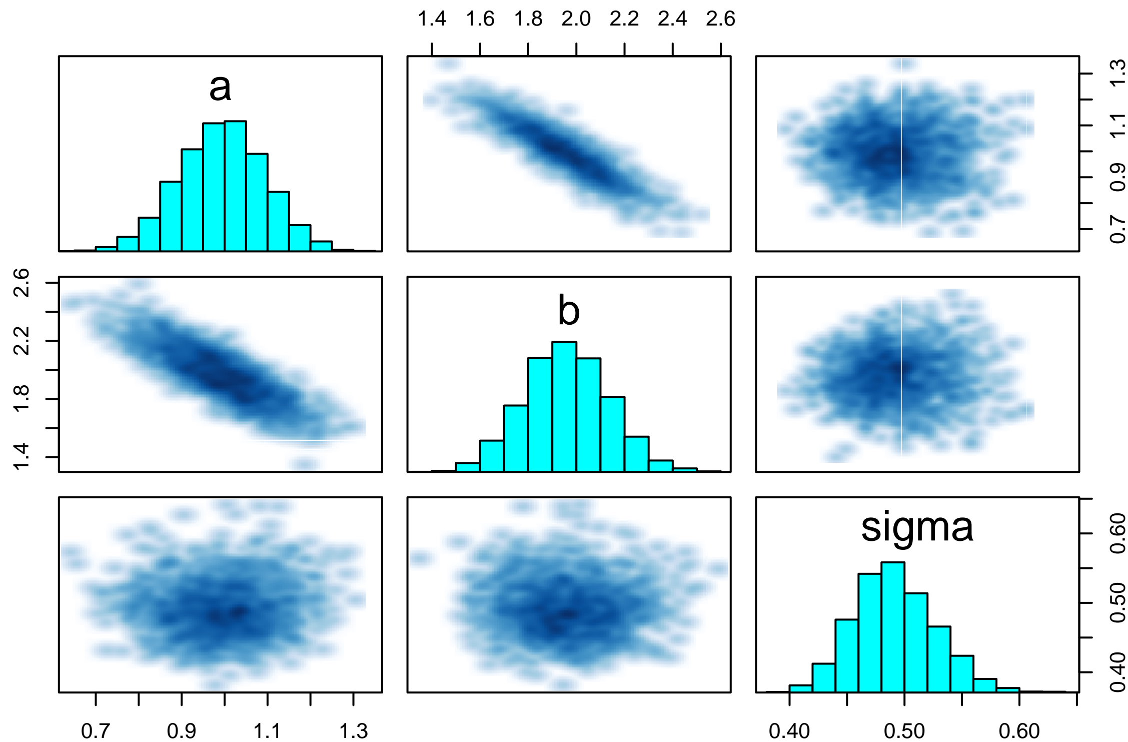


The posterior distribution is multidimensional (here, d=2).

So visualization of the whole posterior is limited, but we can at least look at the pairwise 2d marginals

```r
pairs(fit, pars=c("a","b","sigma"))
```

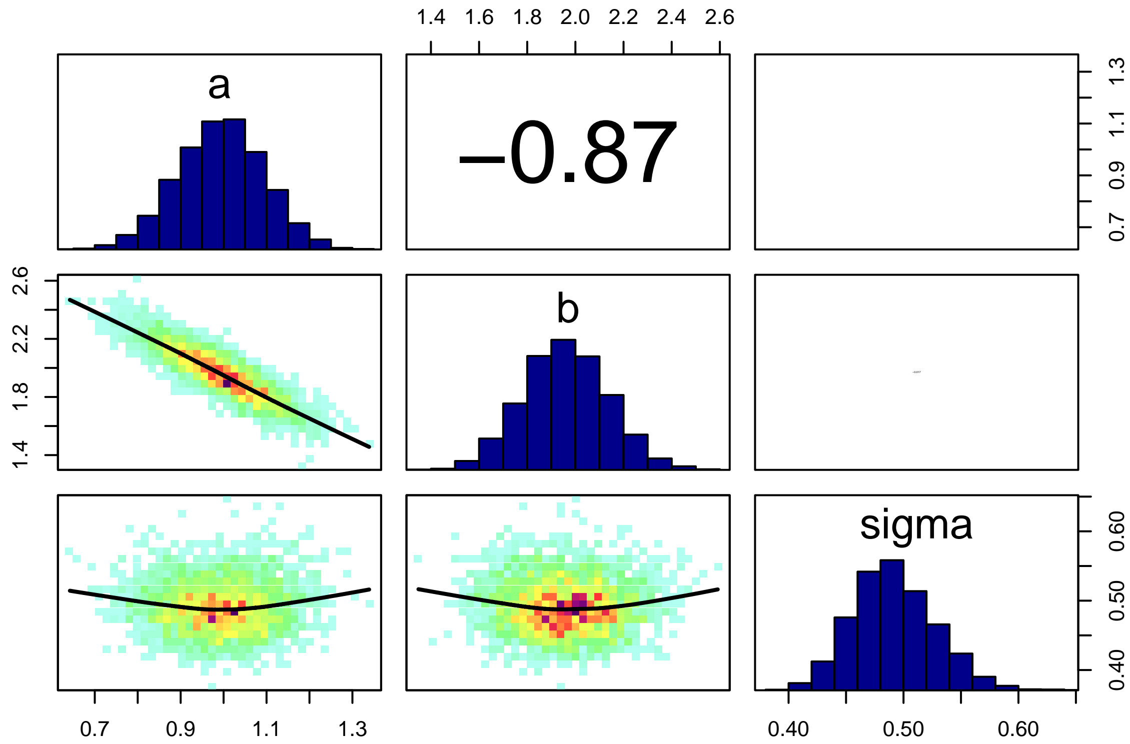
```
## Warning in par(usr): argument 1 does not name a graphical parameter

## Warning in par(usr): argument 1 does not name a graphical parameter

## Warning in par(usr): argument 1 does not name a graphical parameter
```



Cooler pairs plot from `BayesianTools` package, also computes the pairwise correlation coefficients of the parameters

```r
library(BayesianTools)
correlationPlot(as.matrix(fit)[,1:3], thin=1)
```

```
## Warning in par(usr): argument 1 does not name a graphical parameter

## Warning in par(usr): argument 1 does not name a graphical parameter

## Warning in par(usr): argument 1 does not name a graphical parameter

## Warning in par(usr): argument 1 does not name a graphical parameter

## Warning in par(usr): argument 1 does not name a graphical parameter

## Warning in par(usr): argument 1 does not name a graphical parameter
```

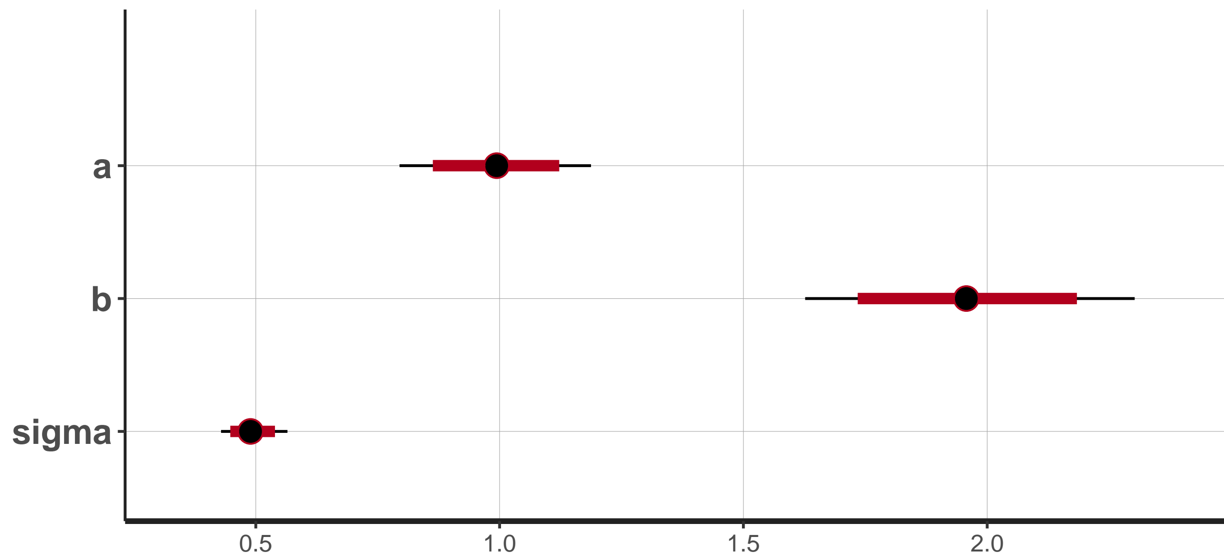


Other Stan plotting options

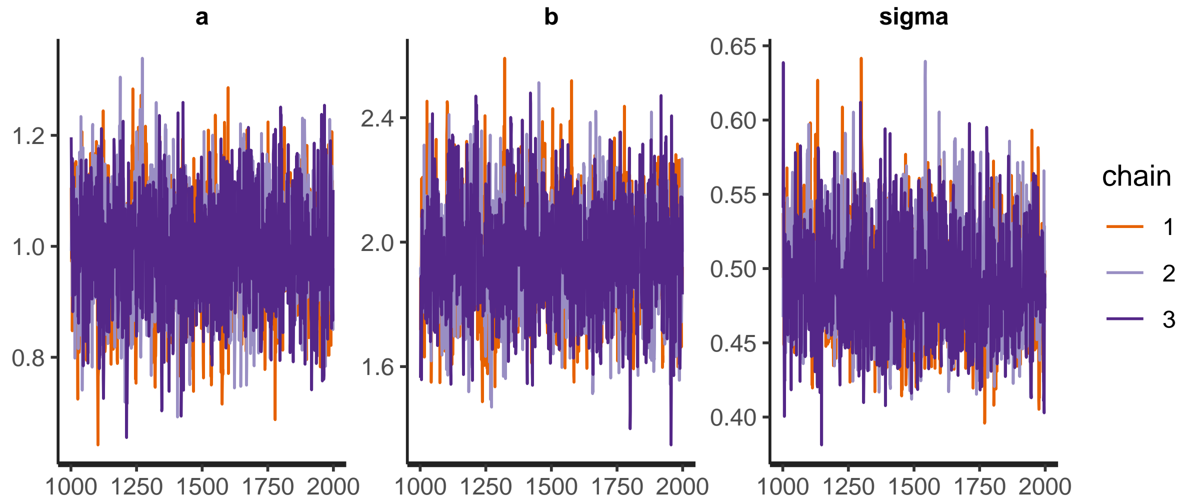
```
stan_plot(fit)
```

```
## ci_level: 0.8 (80% intervals)
```

```
## outer_level: 0.95 (95% intervals)
```

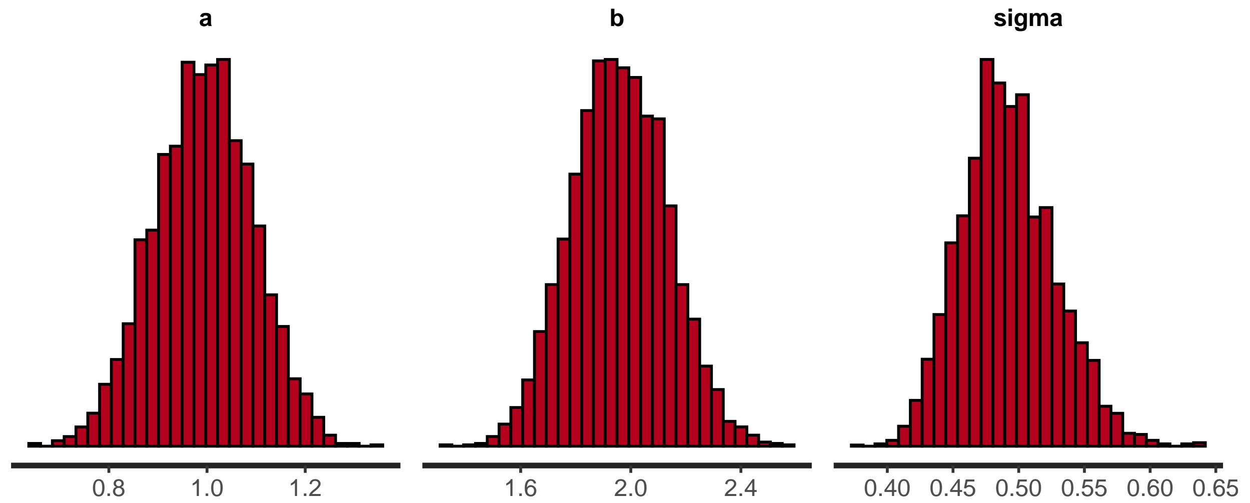


```
stan_trace(fit)
```

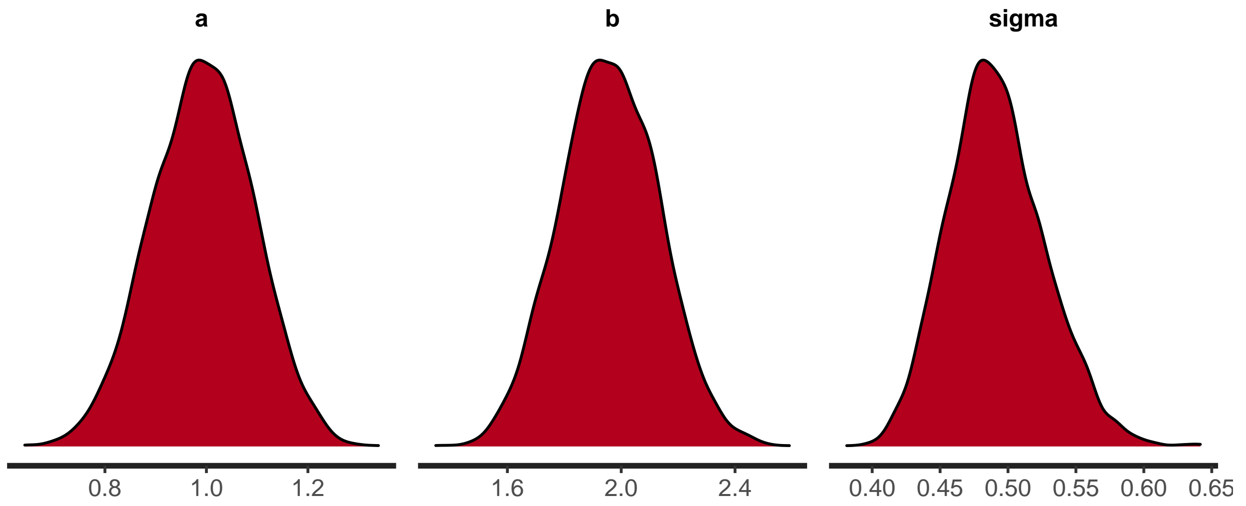



```
stan_hist(fit)
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```




```
stan_dens(fit)
```




```
stan_scat(fit, pars=c("a","b"))
```