Computer Lab 02 — Stan

Contents

1 Introduction **1**

2 Douglas firs example in Stan **2**
2.1 Flat prior 2
2.2 Jeffreys' prior 7

References **12**

1 Introduction

This notebook continues your journey to learn about Stan, another probabilistic programming language. There is plenty of information about the Stan project available at http://mc-stan.org/users/documentation/index.html.

To see a nicely rendered version of this notebook, which is easier to read, you should click on the Preview button. The typeset version of this notebook is also available on LMS so that you can see what output you are supposed to see when running the code. But you are free to modify the code and to experiment.

We will use Stan via R and, in general, will need the following two packages to do so:

```
library(rstan)
```

```
## Loading required package: StanHeaders
```

```
## Loading required package: ggplot2
```

```
## rstan (Version 2.21.2, GitRev: 2e1f913d3ca3)
```

```
## For execution on a local, multicore CPU with excess RAM we recommend calling
## options(mc.cores = parallel::detectCores()).
## To avoid recompilation of unchanged Stan programs, we recommend calling
## rstan_options(auto_write = TRUE)
```

```
library(bayesplot)
```

```
## This is bayesplot version 1.8.0
```

```
## - Online documentation and vignettes at mc-stan.org/bayesplot
```

```
## - bayesplot theme set to bayesplot::theme_default()
```

```
##    * Does _not_ affect other ggplot2 plots
```

```
##    * See ?bayesplot_theme_set for details on theme setting
```

We might as well execute the commands suggested in these start up messages:

```
options(mc.cores = parallel::detectCores())
rstan_options(auto_write = TRUE)
```

2 Douglas firs example in Stan

Here we will revisit the Douglas firs examples from the lectures and repeat the analysis using Stan. According to the user manual, Stan does not have the geometric distribution implemented. However, it has the negative-binomial distribution[1] implemented and the geometric distribution is a special case of the former. Specifically, the negative-binomial distribution, as described in Stan Development Team (2017, Section 52.1, p. 520ff) or Stan Development Team (2021, Section 14.1, p. 107ff) , with parameters $\alpha = 1$ and $\beta = \frac{p}{1-p}$ is identical to the geometric distribution with success parameter p.

First, we create again from the frequency count the data set as it might have been observed. For convenience, we immediately code the observed values as $0, \ldots, 5$ (as if they are realisations from a geometric distribution) instead of $1, \ldots, 6$ (as if they are realisations from a First Success distribution):

```
y.freq <- c(71, 28, 5, 2, 2, 1)
y.obs <- rep(0:5, times = y.freq)
```

2.1 Flat prior

This model is readily implemented in Stan. You know the drill, hit the green right pointing triangle now, and then continue to read below.

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

parameters{
  real<lower=0, upper=1> p;
}

transformed parameters{
  real beta;
  real q;

  beta = p / (1.0 - p) ;
  q = 1.0 - p;
}

model{
  y ~ neg_binomial(1, beta);
  p ~ beta(1,1);
}
```

The code is now more involved than the examples from last week as we have to use several of the allowed blocks. The `data` block declares the observations that we will pass to Stan. In this case the sample size `n` and a vector of integers, called `y`, of length `n`. This vector will hold the observed values from the geometric distribution. Note that we can use the construct `<lower=0>` to tell Stan that `n` must be a non-negative integer and that each entry in the vector `y` must be non-negative. These constraints are checked at run time when you pass the observed data to Stan. Thus, specifying such contraints provides safeguards against wrong data being passed to Stan, good defensive programming practice.

In the `parameters` block we declare the parameter of our model. Here we have only one parameter, namely `p` (for the success probability p), which is a real. Note that we can use the construct `<lower=0, upper=1>`

[1] Note that Stan provides the negative-binomial distribution in various parameterisations Stan Development Team (2021, Sections 14.1–14.4, p. 107ff), for our purposes the one described in Stan Development Team (2021, Section 14.1, p. 107ff) is most convenient to use.

to tell `Stan` that `p` should always be between 0 and 1. `Stan` will ensure that these constraints will not be violated during run time.

In the `transformed parameters` block, we declare two parameters `beta` and `q`. This block is useful to define parameters that are calculated from the "main" parameters of our models, but on which we will not put priors. Here we have to declare `beta` and set it to `p/(1-p)` as the negative binomial distribution is parameterised by this parameter. The declaration of `q` is more for convenience, so that we can also look at the estimate for $q = 1 - p$.

Finally, the `model` block contains the information on the sampling distribution for our observations (given the parameters), and the priors on the parameters. We could have written the specification for the sampling distribution more explicitly as:

```
for (i in 1:n){
  y[i] ~ neg_binomial(1, beta);
}
```

but the `neg_binomial` command is vectorised and so we chose the more compact form of writing the statement. But you should realise that as the program is written, we have implemented an implicit loop over our vector of observations.

Also, the prior specification

```
p ~ beta(1, 1);
```

would not be necessary in this case. If no prior is specified for a parameter, `Stan` will use a uniform prior for that parameter. That uniform prior is, of course, restrained to the parameter space of the parameter. When declaring `p` in the `parameters` block, we specified that the parameter space for this parameter is the interval $[0, 1]$. Thus, in the absence of any other specification of a prior, `Stan` would anyhow use the uniform distribution on $[0, 1]$ as the prior for p.

Hopefully the model is compiled by now. We still have to put the data that we want to pass to `Stan` into a named list, where the left hand side of each `=` is the name of the variable used in the `Stan` program and the right hand side is the name of the object in our `R` session. Finally we call the compiled Stan code via the `sampling()` command:

```
data.in <- list(y = y.obs, n = length(y.obs))
fit1 <- sampling(geom_flat, data = data.in)
```

The simulated values from the posterior should now be in the object `fit1` and we can create some summary statistics for them:

```
print(fit1, pars = c("p", "q"), digits = 5)
```

```
## Inference for Stan model: b5f488323a4e84cedf74f042a072a554.
## 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    2.5%     25%     50%     75%   97.5% n_eff    Rhat
## p 0.65481 0.00093 0.03648 0.57978 0.63080 0.65546 0.67952 0.72372  1530 0.99992
## q 0.34519 0.00093 0.03648 0.27628 0.32048 0.34454 0.36920 0.42022  1530 0.99992
##
## Samples were drawn using NUTS(diag_e) at Fri Mar 12 16:27:03 2021.
## 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).
```

How do these values compare with those obtained in `OpenBUGS`?

`Stan` is using a Hamiltonian Markov Chain (HMC) sampler. So it is a good idea to always change the diagnostics, more about them in (later) lectures, provided by the `rstan` package:

```
check_hmc_diagnostics(fit1)
```

```
##
## Divergences:
```

```
## 0 of 4000 iterations ended with a divergence.
```

```
##
## Tree depth:
```

```
## 0 of 4000 iterations saturated the maximum tree depth of 10.
```

```
##
## Energy:
```

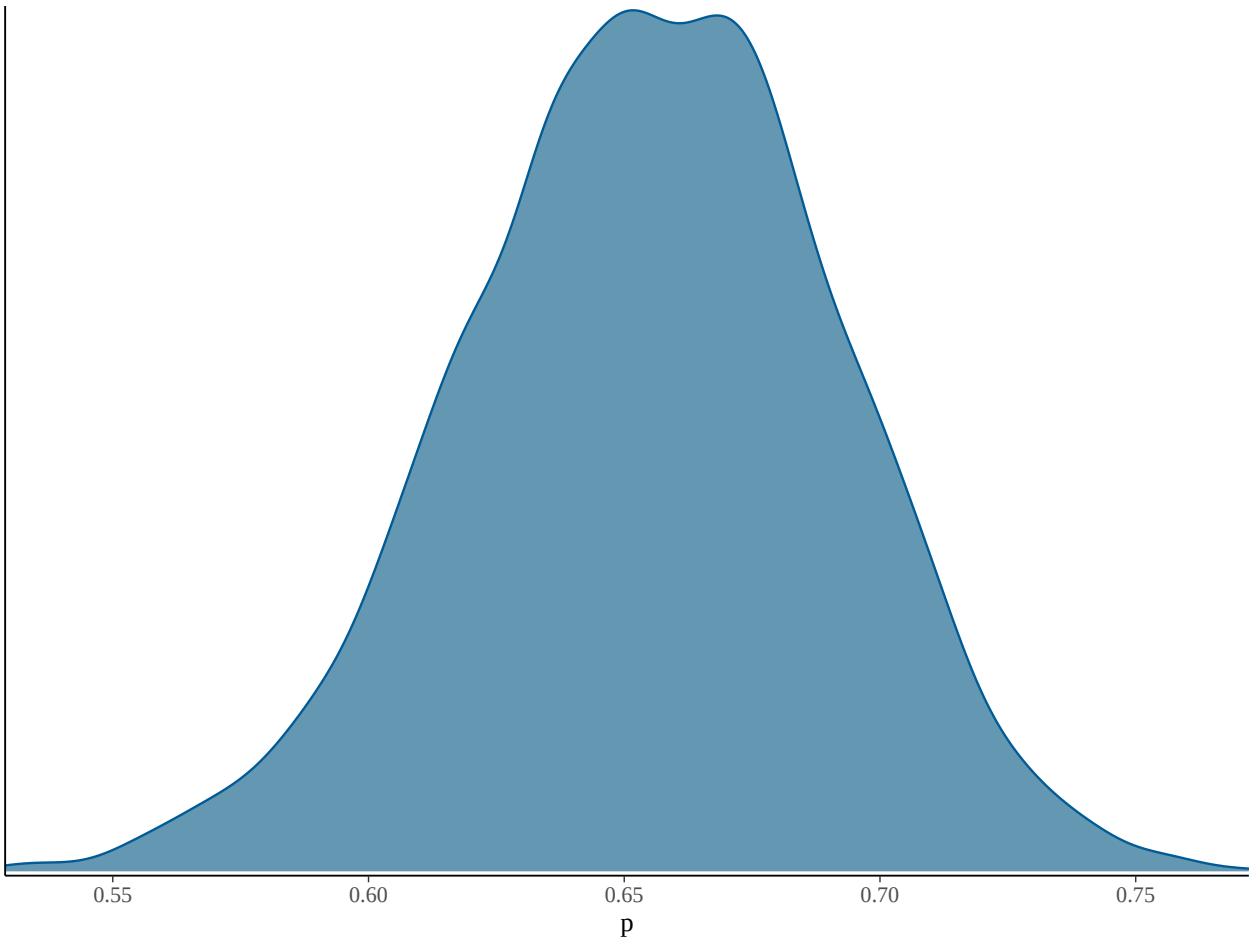
```
## E-BFMI indicated no pathological behavior.
```

We can also use commands from the `bayesplot` package to make some pretty pictures. First we extract the sampled values from the fitted model object:

```
posterior <- as.array(fit1)
```

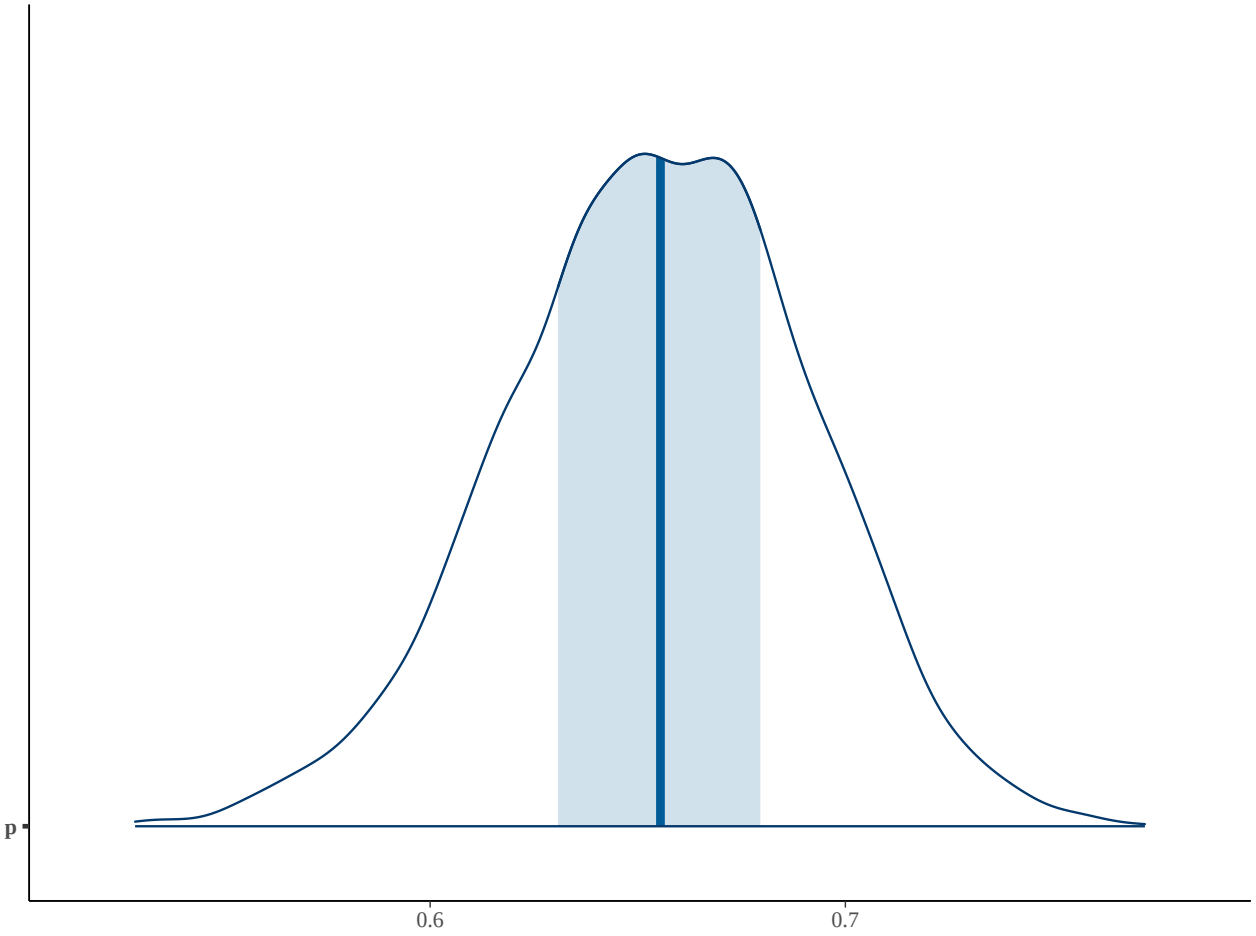
A plot that shows the posterior density of p:

```
mcmc_dens(posterior, pars = "p")
```



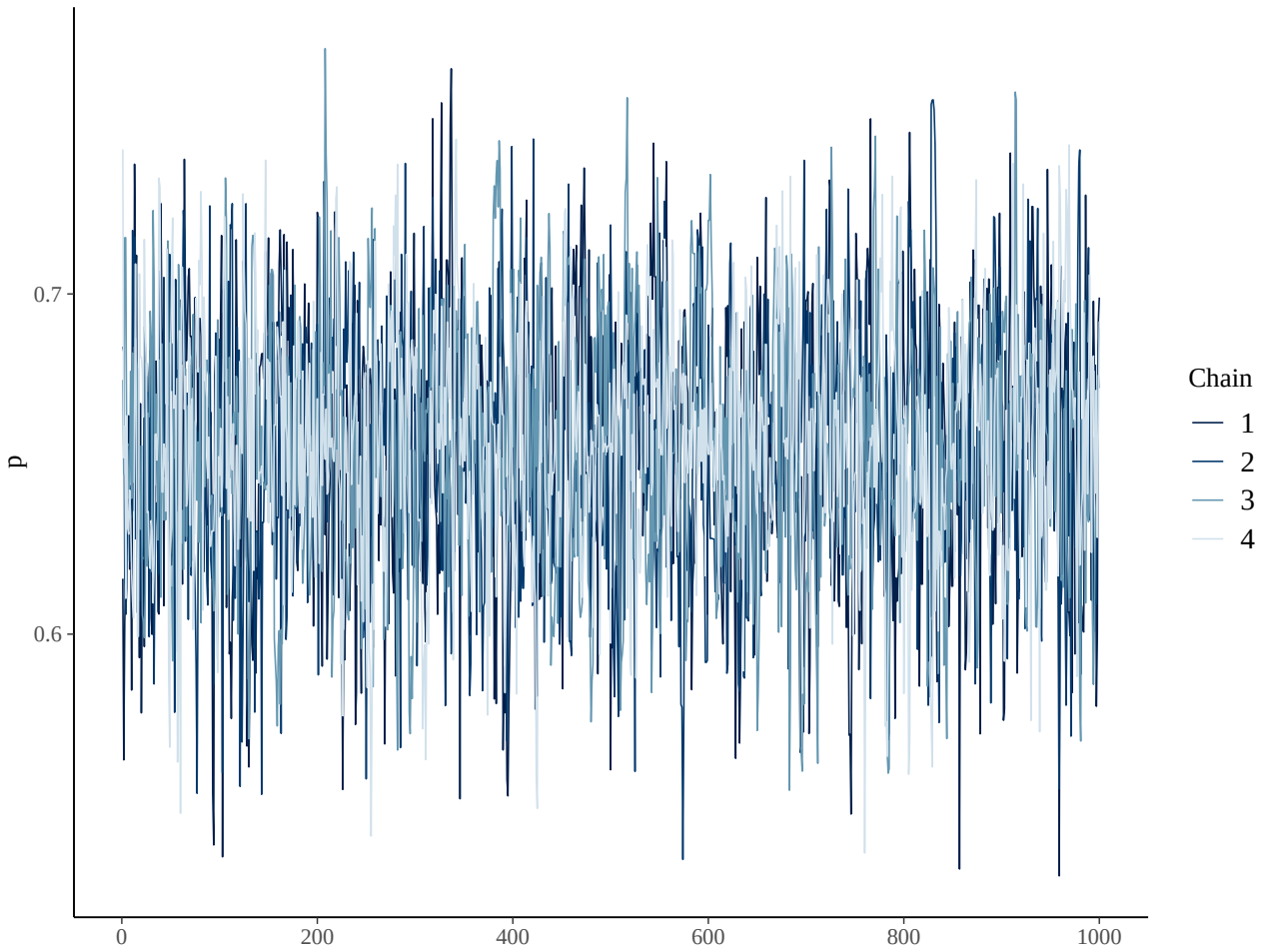
A plot that shows the posterior density of p and highlights a "central" area in which `prob` of the probability mass is. Experiment with how this plot changes when you vary `prob`:

```
mcmc_areas(posterior, pars = "p", prob = 0.5)
```



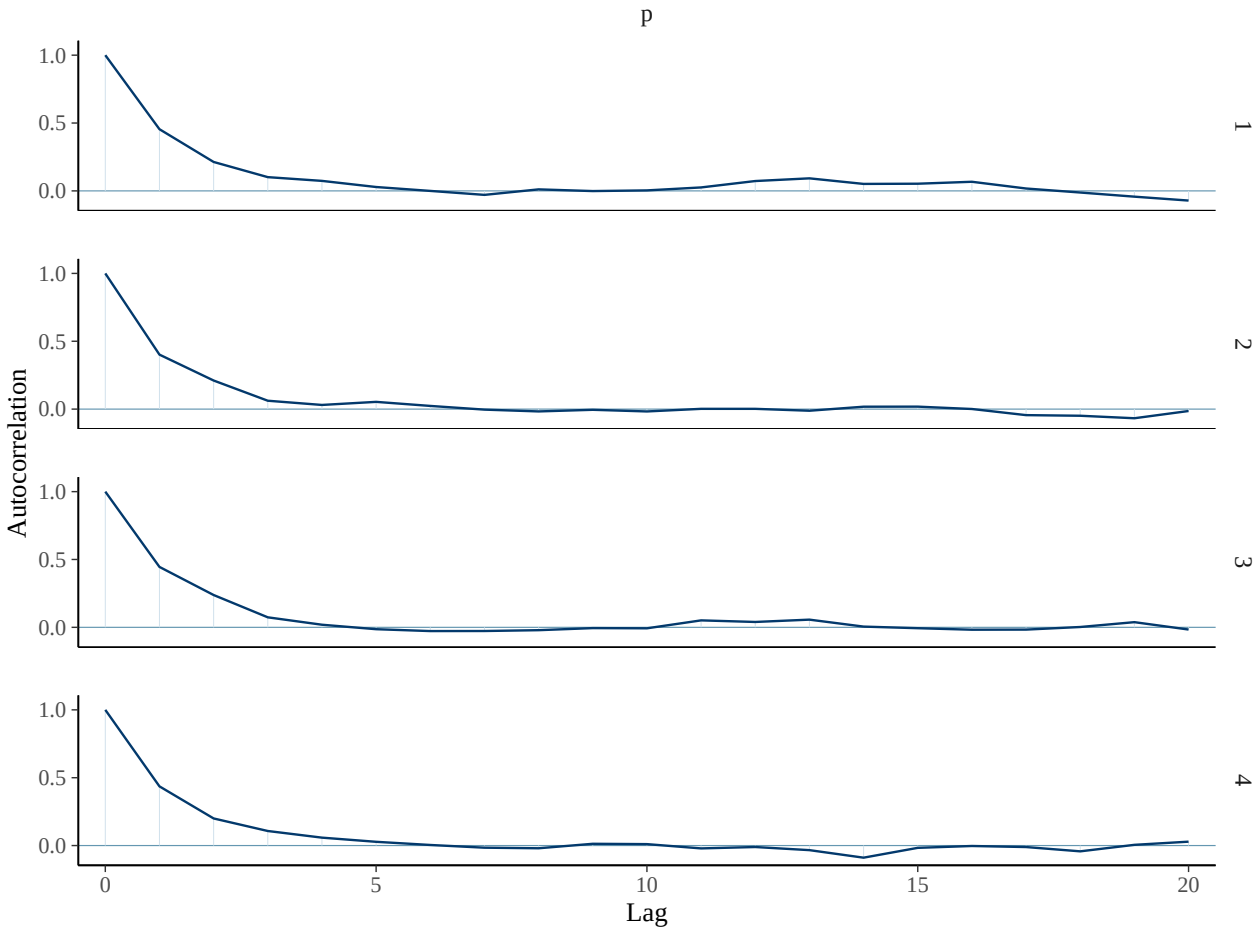
A plot of the traces for the four chains:

```
mcmc_trace(posterior, pars = "p")
```



And a plot of the estimated auto-correlation function for each chain:

```
mcmc_acf(posterior, pars = "p")
```



2.2 Jeffreys' prior

To use Jeffreys' prior, we only have to make minor changes to our program. You know the drill, hit the green right pointing triangle now, and then continue to read below.

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

parameters{
  real<lower=0> u;
}

transformed parameters{
  real beta;
  real p;
  real q;

  p = 1 - ( (1-exp(-u))/(1+exp(-u)) )^2;
  beta = p / (1.0 - p);
  q = 1.0 - p;
}

model{
```

```
  y ~ neg_binomial(1, beta);
}
```

The `data` block remains the same, naturally.

Our `parameters` block changes. Now our main parameter is `u` (u) which will have a flat distribution (uniform prior) on the non-negative real line. So we declare this parameter as a `real` and specify that it must be non-negative via a `<lower=0>` construct.

Our `tranformed parameters` block changes, as we now declare `p` (p) in this block and calculate it from `u` (u). Once we know `p`, we can calculate our other transformed parameters `beta` and `q` as before.

Finally, the `model` block does not really change. We still specify the sampling distribution for our observed data (given the parameters) in a vectorised manner. As we want to have a flat prior on `u` (u), but `Stan` does not allow to *specify* improper priors, we just do not put a prior at all on `u`; in this manner `Stan` will automatically use the improper uniform prior for `u`.

Hopefully the model is compiled by now. If so, we call the compiled Stan code via the `sampling()` command:

```
fit2 <- sampling(geom_jeffreys, data = data.in)
```

The simulated values from the posterior should now be in the object `fit2` and we can create some summary statistics for them:

```
print(fit2, pars = c("p", "q"), digits = 5)
```

```
## Inference for Stan model: 97e380154c5f148ae2435efa605536bd.
## 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    2.5%     25%     50%     75%   97.5% n_eff    Rhat
## p 0.65464 0.00097 0.03715 0.58079 0.63052 0.65457 0.67968 0.72597  1468 0.99979
## q 0.34536 0.00097 0.03715 0.27403 0.32032 0.34543 0.36948 0.41921  1468 0.99979
##
## Samples were drawn using NUTS(diag_e) at Fri Mar 12 16:27:41 2021.
## 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).
```

How do these values compare with those obtained in `OpenBUGS`?

`Stan` is using a Hamiltonian Markov Chain (HMC) sampler. So it is a good idea to always change the diagnostics, more about them in (later) lectures, provided by the `rstan` package:

```
check_hmc_diagnostics(fit2)
```

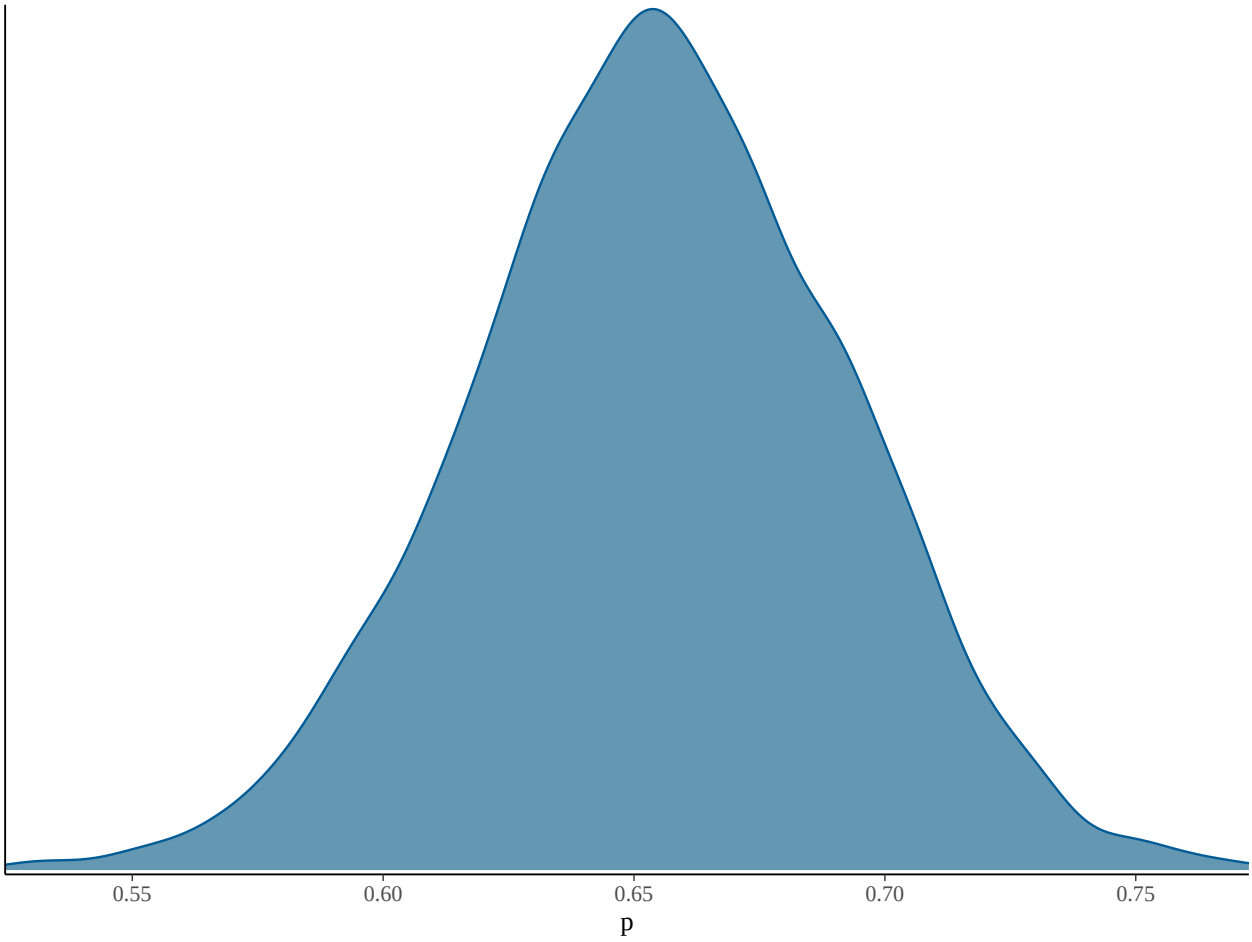
```
##
## Divergences:
## 0 of 4000 iterations ended with a divergence.
##
## Tree depth:
## 0 of 4000 iterations saturated the maximum tree depth of 10.
##
## Energy:
## E-BFMI indicated no pathological behavior.
```

We can also use commands from the `bayesplot` package to make some pretty pictures. First we extract the sampled values from the fitted model object:

```
posterior <- as.array(fit2)
```

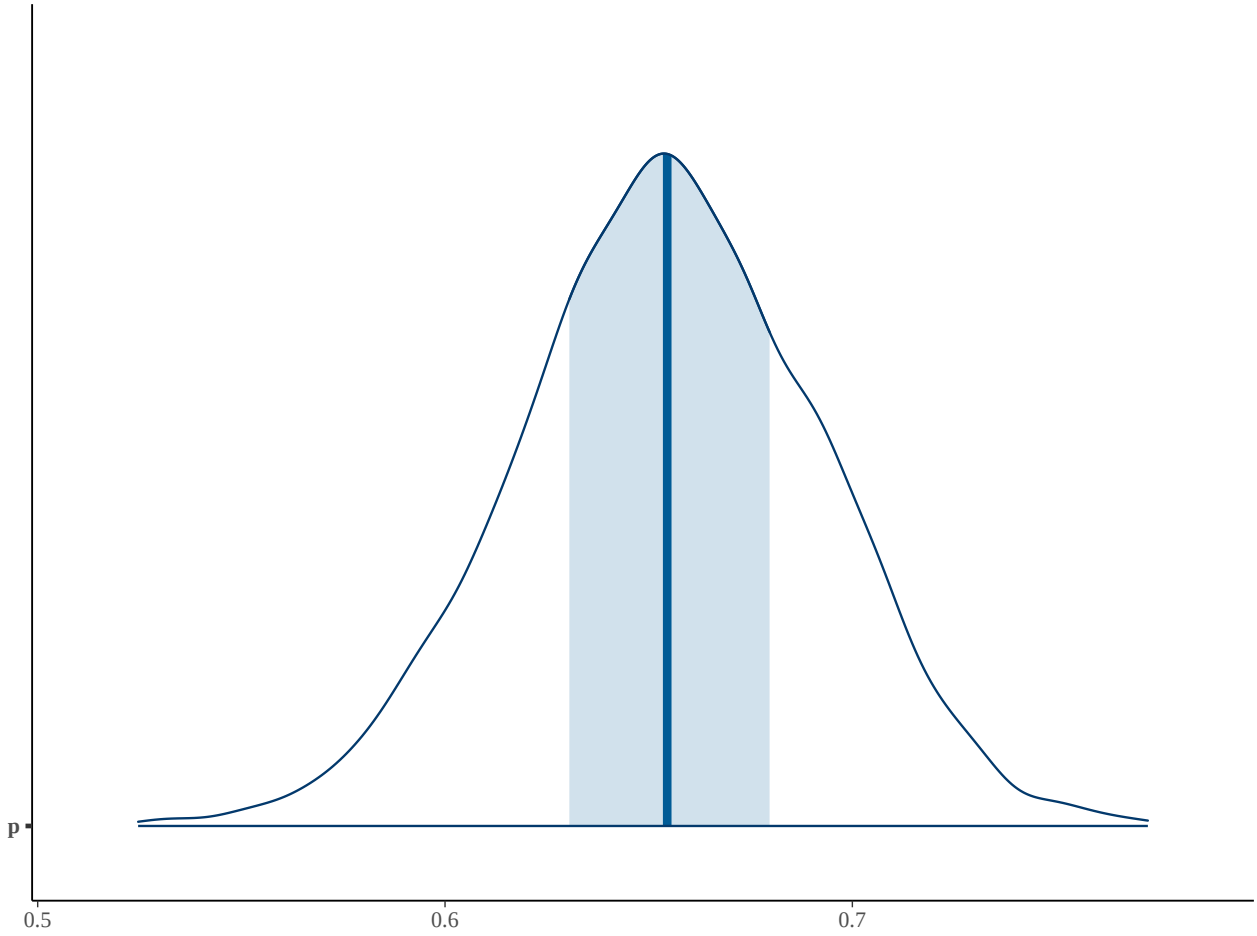
A plot that shows the posterior density of p:

```
mcmc_dens(posterior, pars = "p")
```



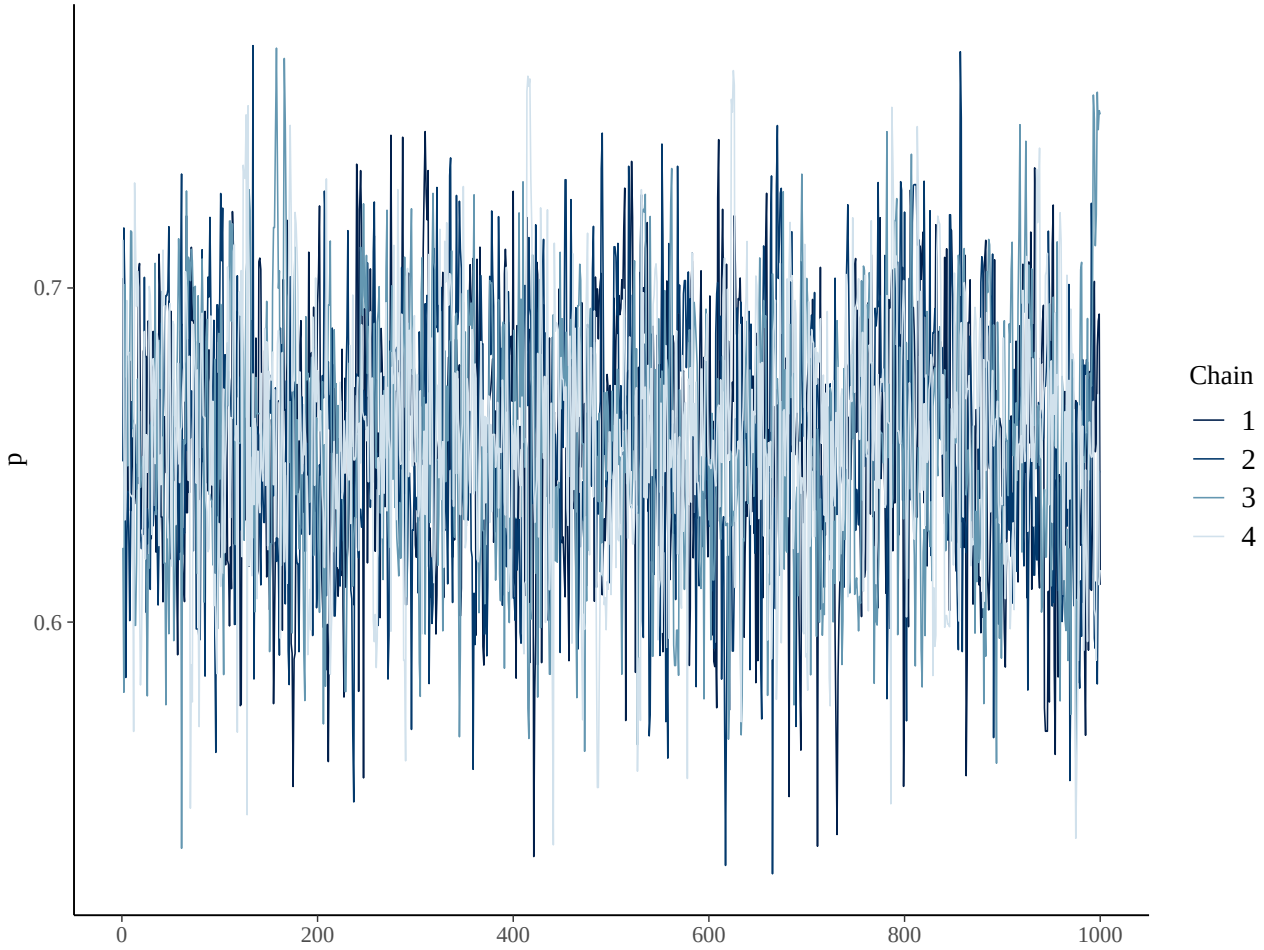
A plot that shows the posterior density of p and highlights a "central" area in which `prob` of the probability mass is. Experiment with how this plot changes when you vary `prob`:

```
mcmc_areas(posterior, pars = "p", prob = 0.5)
```



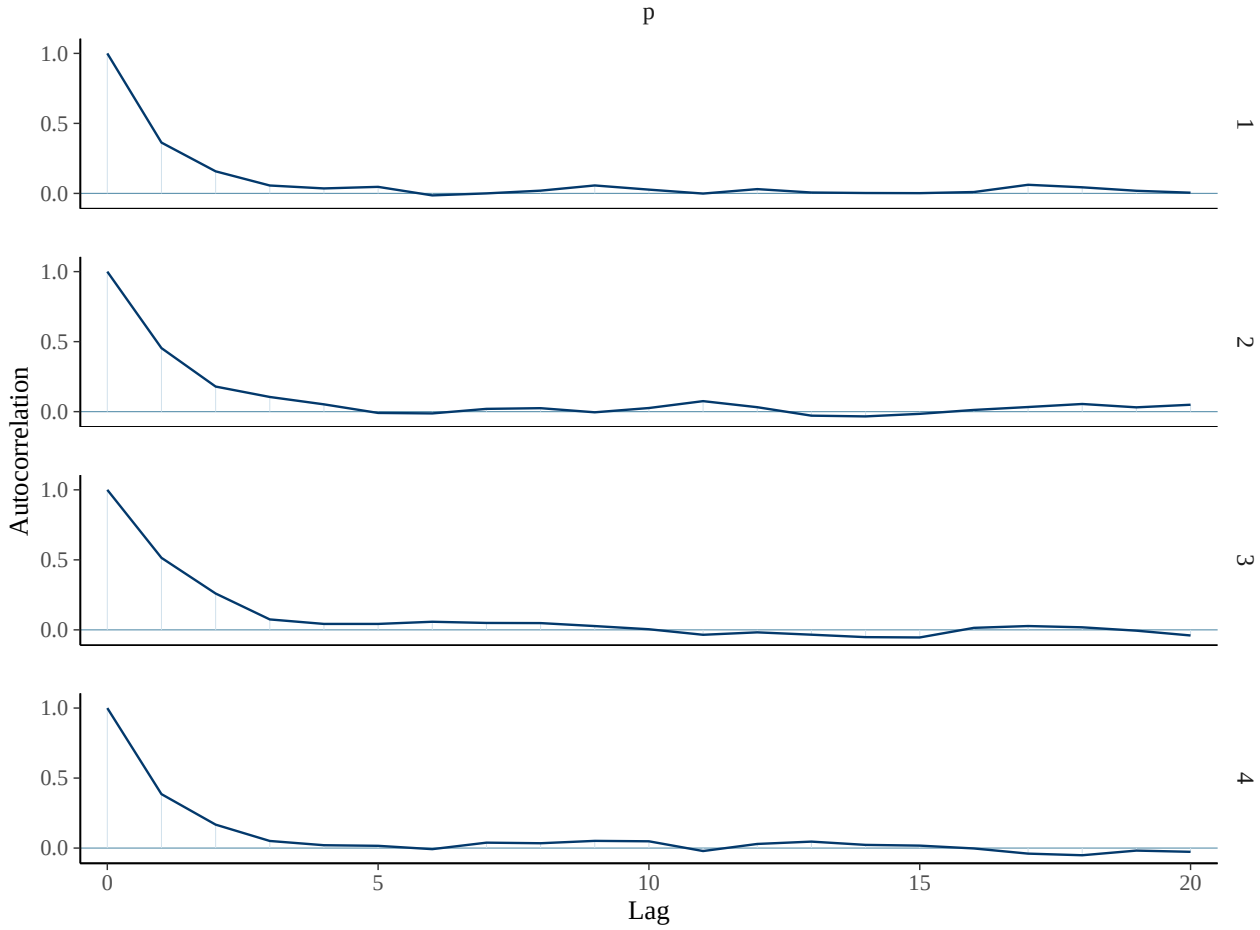
A plot of the traces for the four chains:

```
mcmc_trace(posterior, pars = "p")
```



And a plot of the estimated auto-correlation function for each chain:

```
mcmc_acf(posterior, pars = "p")
```



References

Stan Development Team. 2017. *Stan Modeling Language: User's Guide and Reference Manual*. Version 2.17.1. http://mc-stan.org.

———. 2021. *Stan Functions Reference*. Version 2.26. http://mc-stan.org.