Bayesian Inference for SIR Epidemic Models using odin and monty

Contents

Introduction **1**
Why odin and mcstate? . . . 2
The SIR Model (Quick Review) . . . 2

Housekeeping (load required libraries and set seed for reproducibility) **2**

Step 1: Simulate SIR Epidemic Data **3**

Step 1: Simple model run **3**

Step 2: Fitting **6**

Step 3: The MCMC **7**
3.1 Fit with a random walk . . . 8
3.2 Fit with an adaptive Metropolis-Hastings sampler . . . 9

Step 4: Check results **9**

Step 5: Convergence diagnostics **9**

Step 6: Tuning the MCMC (optional) **20**

Step 7: Compare the estimates to the true values **23**

Author: Julia Mayer

Introduction

This notebook exemplifies the use of Bayesian statistics for **SIR (Susceptible-Infected-Recovered) epidemic models** using the **odin, dust, and monty** packages. The packages, unlike BayesianTools, provides a "domain specific language" (DSL) which looks like R but is compiled directly to C++. This makes them very fast while not requiring the knowledge of other languages such as Stan or C++. **odin2** is used to define the SIR model by specifying the ordinary differential equations that govern it. **dust** compiles the model to a particle filter-compatible format, and **monty** performs the MCMC fitting.

We'll cover:

1. Understanding the SIR model (same as before!)
2. Simulating epidemic data
3. Defining likelihood and prior functions
4. Fitting with various MCMC algorithms (Random Walk, Metropolis-Hastings)
5. Trace plots for convergence diagnostics
6. Posterior predictive checks
7. Tuning the MCMC samplers

For more information on the odin, odin.dust, and mcstate packages see:

- odin2: https://mrc-ide.github.io/odin2/
- dust2: https://mrc-ide.github.io/dust2/
- monty: https://mrc-ide.github.io/monty/

This notebook is heavily inspired by the **monty** and **bayesintro** vignettes.

Why odin and mcstate?

Advantages:

- R interface - no experience with Stan, C++ or C needed
- Multiple MCMC algorithms available
- Faster than BayesianTools

Trade-offs:

- Steep learning curve
- Very new packages so limited flexibility
- Not all functions are supported in odin models, sometimes making debugging difficult
- Limited community support compared to more established packages like Bayesiantool
- No built-in diagnostic tools

The SIR Model (Quick Review)

The SIR model divides a population into three compartments:

$$\begin{aligned}\frac{dS}{dt} &= -\beta\frac{SI}{N} \\ \frac{dI}{dt} &= \beta\frac{SI}{N} - \gamma I \\ \frac{dR}{dt} &= \gamma I\end{aligned}$$

Where:

- S: Number of susceptible individuals
- I: Number of infected individuals
- R: Number of recovered individuals
- N: Total population size ($N = S + I + R$)
- β: Transmission rate
- γ: Recovery rate
- $\frac{1}{\gamma}$: Average infectious period
- $R_0 = \frac{\beta}{\gamma}$: Basic reproduction number
- I_0: Initial number of infected individuals

Housekeeping (load required libraries and set seed for reproducibility)

```
set.seed(42)

if (!require("pacman")) install.packages("pacman")
```

```
## Loading required package: pacman
```

```
pacman::p_load(odin2, dust2, monty, dplyr, tidyr, ggplot2, kableExtra, knitr,
               bayesplot, posterior)
```

Step 1: Simulate SIR Epidemic Data

We'll use an SIR model to simulate incidence data. You can choose between a deterministic and a stochastic version of the SIR model, which are written in separate files for readability. Head to the files to see the details of the models. You will see some comments there explaining the different parts of the model.

```
# Load model
# change the path to where your model is saved
model_path <- "/Users/juliamayer/Library/CloudStorage/OneDrive-Charité-UniversitätsmedizinBerlin/Traini
model_file <- paste0(model_path, "deterministic SIR odin.R")
# model_file <- paste0(model_path, "stochastic SIR odin.R")

gen_sir <- odin(model_file)
```

```
## -- R CMD INSTALL ---------------------------------------------------------
## * installing *source* package 'odin.system6be87ddd' ...
## ** using staged installation
## ** libs
## using C++ compiler: 'Apple clang version 17.0.0 (clang-1700.4.4.1)'
## using SDK: 'MacOSX26.1.sdk'
## clang++ -arch arm64 -std=gnu++17 -I"/Library/Frameworks/R.framework/Resources/include" -DNDEBUG  -I'
## clang++ -arch arm64 -std=gnu++17 -I"/Library/Frameworks/R.framework/Resources/include" -DNDEBUG  -I'
## clang++ -arch arm64 -std=gnu++17 -dynamiclib -Wl,-headerpad_max_install_names -undefined dynamic_loo
## installing to /private/var/folders/2y/bb_ym4f535j0plsxk3hbxqxm0000gn/T/RtmpXPwYfW/devtools_install_4
## ** checking absolute paths in shared objects and dynamic libraries
## * DONE (odin.system6be87ddd)
```

Step 1: Simple model run

This can be helpful to check that your model is behaving as expect (e.g., no negative numbers for people). We will also use to generate data to fit the model to later. We set the true values of the parameters we will later try to estimate. We set the initial number of infected individuals to 1, the transmission rate β to 0.1 and the average infectious period to 20 days ($\gamma = 0.05$). With these values, we have $R_0 = \frac{\beta}{\gamma} = 2$.

Here, we are generating incidence data,

```
# You can parse parameter values and initial states that are different from the
# ones specified in your odin model
true_pars <- list(beta = 0.1,
             gamma = 0.05,
             I0 = 1
)

# You can try different parameter values here
# pars <- list(beta = 0.1,
#              gamma = 0.03,
#              I0 = 2
# )

# View parameters
kable(as.data.frame(true_pars),
```

```
      caption = "True parameter values", align = "cc") |>
  column_spec(1:3, width = "13mm")
```

Table 1: True parameter values

beta	gamma	I0
0.1	0.05	1

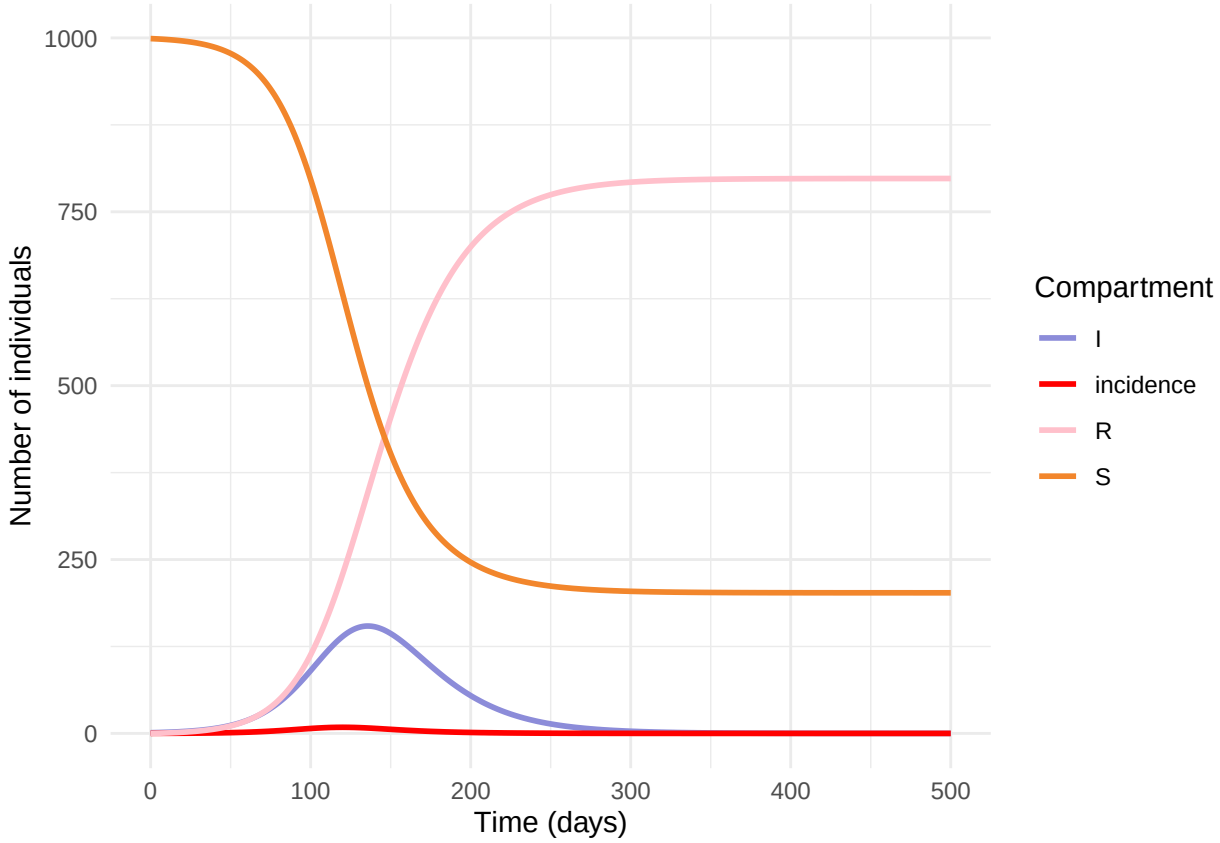
We need to create a dust system from our *odin* model in order to simulate it. You can choose how many steps per day you want to take with the *dt* argument and how many runs you want to simulate with *n_particles*. All systems start with a zero state unless you set them via *dust_system_set_state()* or *dust_system_set_state_initial()*.

In case you're using a stochastic model, each run will produce a different result. Since, we're using the model to produce data here, we want to make sure that we get a clear epidemic curve so we run the model until we get a peak of at least 50 infected individuals.

```
max_inf <- 0

while (max_inf < 50){
  # We use 1 particle here as we are just running one simulation.
  # With dt = 0.25, we take 4 steps per day. This can be changed to 0.5 or 1.
  sys <- dust_system_create(gen_sir(), true_pars, n_particles = 1, dt = 0.25)
  dust_system_set_state_initial(sys)

  # This will run one iteration of the model forward over 500 days.
  time <- 0:500

  y <- dust_system_simulate(sys, time)
  max_inf <- max(dust_unpack_state(sys, y)$I)
}
```

```
# Visualise how the population changes over time
# Define a palette
palette <- c("S" = "#F2862C",  "I" = "#8c8cd9", "R" = "pink", incidence = "red")

# Extract the values of the different compartments over time
S <- dust_unpack_state(sys, y)$S
I <- dust_unpack_state(sys, y)$I
R <- dust_unpack_state(sys, y)$R
cases <- dust_unpack_state(sys, y)$incidence

# Combine them in a dataframe for easier plotting
states <- data.frame(time = time,
                     S = S,
                     I = I,
                     R = R,
                     incidence = cases)

# Plot
states %>%
  pivot_longer(cols = -time, names_to = "Compartment", values_to = "individuals") %>%
  ggplot(aes(x = time, y = individuals, color = Compartment)) +
  geom_line(linewidth = 1) +
```

```
  scale_color_manual(values = palette) +
  labs(x = "Time (days)", y = "Number of individuals") +
  theme_minimal()
```



We will extract the number of cases at specific time points to generate incidence data.

```
# Extract the number of cases at specific time points
time_points <- c(30, 60, 90, 120, 150, 180, 240, 300, 360, 420, 480)
incidence <- data.frame(time = time_points,
                        cases = round(cases[time_points + 1]))
# +1 because time starts at 0

# View the incidence data
kable(incidence, col.names = c("Time (days)", "Incidence"),
             caption = "Simulated incidence data", align = "cc") |>
  column_spec(1:2, width = "40mm")
```

Table 2: Simulated incidence data

Time (days)	Incidence
30	0
60	2
90	5
120	9
150	6
180	3
240	0

300	0
360	0
420	0
480	0

Step 2: Fitting

If the model is behaving as expected, you can move on to the fitting. We will fit the model to the generated incidence data using the *monty* package.

To fit the model, we need a function that tells us if the model output is close to the observed data. This is the comparison function that is used by the particle filter and that computes the log-likelihood. When using *odin2* and *monty*, this is done in the model (see the last three lines of the model code). We can now build a particle filter that will use this function and test the outputs.

```
# Build a filter and test it using the true parameter values
filter <- dust_filter_create(gen_sir,
                             time_start = 0,
                             data = incidence,
                             n_particles = 1000) # we run the filter 1,000 times

dust_likelihood_run(filter,
                    pars = true_pars,
                    save_trajectories = T) # save the trajectories for plotting
```

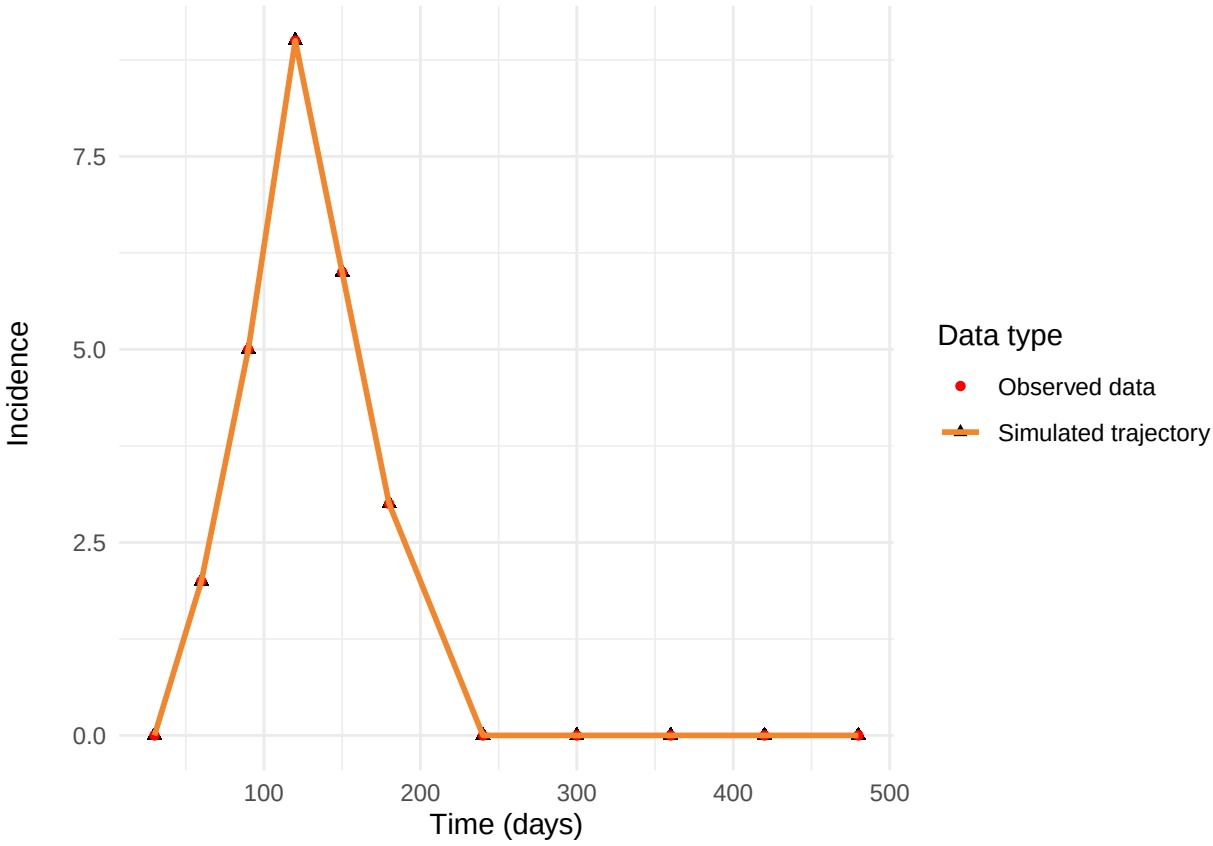
```
## [1] -9.35598
```

```
# You can try different values here to see how the likelihood changes
# dust_likelihood_run(filter,
#                      pars = list(beta = 0.1,
#                                  gamma = 0.03,
#                                  I0 = 45)
#                      )
```

When using real data it can be helpful to plot the model and the data together to see why the filter is returning a specific likelihood value, especially if it's not what you would expect (e.g., it's very low).

```
# Plot the fit
# You can try doing this for different values to see how the model fits the data
h <- dust_likelihood_last_trajectories(filter)

# We extract the values of the 1,000 runs
traj <- data.frame((dust_unpack_state(filter, h)$incidence)) %>%
  mutate(run_id = row_number()) %>%
  pivot_longer(cols = -run_id, names_to = "time", values_to = "cases") %>%
  mutate(time = as.integer(gsub("X", "", time)),
         cases = round(cases)) %>%
  merge(data.frame(time_points) %>%
          mutate(time = row_number()), by = "time") %>%
  arrange(run_id, time)

# We take the median of the simulated trajectories since we simulated 1,000 runs
traj_med <- traj %>%
  group_by(time_points) %>%
  summarise(cases = median(cases))
```

```
traj %>% ggplot() +
  geom_point(aes(x = time_points, y = cases, shape = "Simulated trajectory")) +
  geom_point(data = incidence,
             aes(x = time, y = cases,
                 shape = "Observed data", col = "Observed data")) +
  geom_line(data = traj_med,
            aes(x = time_points, y = cases, col = "Simulated trajectory"),
            linewidth = 1) +
  scale_color_manual(values = c("Simulated trajectory" = "#F2862C",
                                "Observed data" = "red")) +
  labs(x = "Time (days)",
       y = " Incidence\n",
       shape = "Data type",
       colour = "Data type") +
  theme_minimal()
```



If you're using the stochastic model, each trajectory is different but the median values (orange line) capture the shape of the observed data. If you're using a deterministic model, all trajectories will be the same and the simulated data should be equal to the observed data.

Step 3: The MCMC

Now we can set up our MCMC. *monty* offers 4 different samplers:

- Random Walk Sampler
- Adaptive Metropolis-Hastings Sampler

- Hamiltonian Monte Carlo
- Parallel Tempering Sampler

```
# The first step for any of these samplers is to define a list of inputs
# We fix the value of the I0 to the true value
packer <- monty_packer(c("beta", "gamma"), fixed = list(I0 = 1))

# Then we define the likelihood function ...
likelihood <- dust_likelihood_monty(filter, packer, save_trajectories = T)

# ... the priors
prior <- monty_dsl({
  beta ~ Exponential(mean = 0.3)
  gamma ~ Exponential(mean = 0.1)
  # You can add all the other parameters you want to estimate (e.g. I0) here
})

# ... and the posterior distribution
posterior <- likelihood + prior

# The samplers need a variance-covariance matrix to define the step sizes
# for each parameter. This should be an YxY diagonal matrix where Y is the
# number of parameters you're estimating
vcv <- diag(2) * 0.02 # We set up a naive one with a small variance

# Finally, we need to define the model properties
# If using the stochastic model, comment the next 2 lines and un-comment the third
properties_deter <- monty_model_properties(is_stochastic = F)
model <- monty_model(posterior, properties = properties_deter)
# model <- monty_model(posterior)
```

Finally, we need to choose our sampler. We will experiment with the random walk sampler and the adaptive Metropolis-Hastings sampler.

3.1 Fit with a random walk

```
sampler_rw <- monty_sampler_random_walk(vcv)

# You can parallelise the process
runner <- monty_runner_callr(2, progress = T)
# this should be a multiple of the number of chains you're running

# And we're ready to sample (this takes about 6 minutes on my laptop for the
# stochastic model and about 2 minutes for the deterministic one)
samples_rw <- monty_sample(model,
                           sampler_rw,
                           20000, # We run 20000 iterations
                           initial = packer$pack(true_pars),
                           # These are our initial values. They should be in the
                           # same order as the parameter names in the packer above
                           n_chains = 2, # We run 2 chains
                           runner = runner)
```

3.2 Fit with an adaptive Metropolis-Hastings sampler

```
sampler_mh <- monty_sampler_adaptive(vcv)

# This takes about about 2 minutes on my laptop for the deterministic model and
# 8 minutes for the stochastic model
samples_mh <- monty_sample(model,
                           sampler_mh,
                           20000, # We run 20000 iterations
                           initial = packer$pack(true_pars),
                           # These are our initial values. They should be in the
                           # same order as the parameter names in the packer above
                           n_chains = 2, # We run 2 chains
                           runner = runner)
```

Step 4: Check results

We can check our results. These tools must be imported from other packages. We use *posterior* here.

```
draws_rw <- as_draws_df(samples_rw)
summarise_draws(draws_rw)
```

```
## # A tibble: 2 x 10
##   variable   mean median     sd    mad      q5    q95  rhat ess_bulk ess_tail
##   <chr>     <dbl>  <dbl>  <dbl>  <dbl>   <dbl>  <dbl> <dbl>    <dbl>    <dbl>
## 1 beta     0.0969 0.0966 0.0220 0.0219 0.0620  0.135   1.01     503.     451.
## 2 gamma    0.0455 0.0445 0.0242 0.0246 0.00698 0.0889  1.01     506.     521.
```

```
# This provides the following information:
#   - Empirical (sample) mean
#   - Empirical (sample) median
#   - Empirical (sample) standard deviation
#   - Mean absolute deviation (MAD): "Naive" standard error, that is the standard error of the mean
#     (adjusting for sample size).
#   - Quantiles for each variable
#   - R hat (should be < 1.1)
#   - Effective sample size (the larger the better)
```

```
# Repeat this here for the MH algortihm
draws_mh <- as_draws_df(samples_mh)
summarise_draws(draws_mh)
```

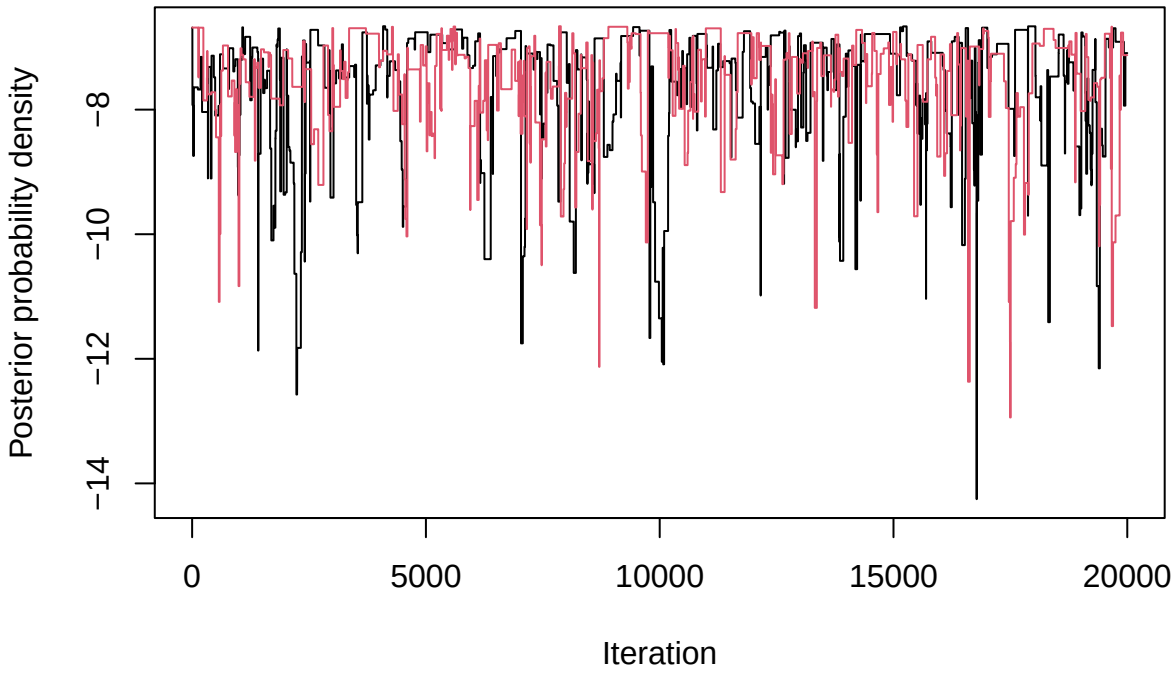
```
## # A tibble: 2 x 10
##   variable   mean median     sd    mad      q5    q95  rhat ess_bulk ess_tail
##   <chr>     <dbl>  <dbl>  <dbl>  <dbl>   <dbl>  <dbl> <dbl>    <dbl>    <dbl>
## 1 beta     0.0953 0.0957 0.0210 0.0218 0.0607  0.130   1.00     494.    1210.
## 2 gamma    0.0439 0.0443 0.0233 0.0247 0.00567 0.0826  1.00     487.    1156.
```

You can already see that the estimates are pretty good. If they weren't, this could be due to poor mixing of the chains. We will check this in the next step.

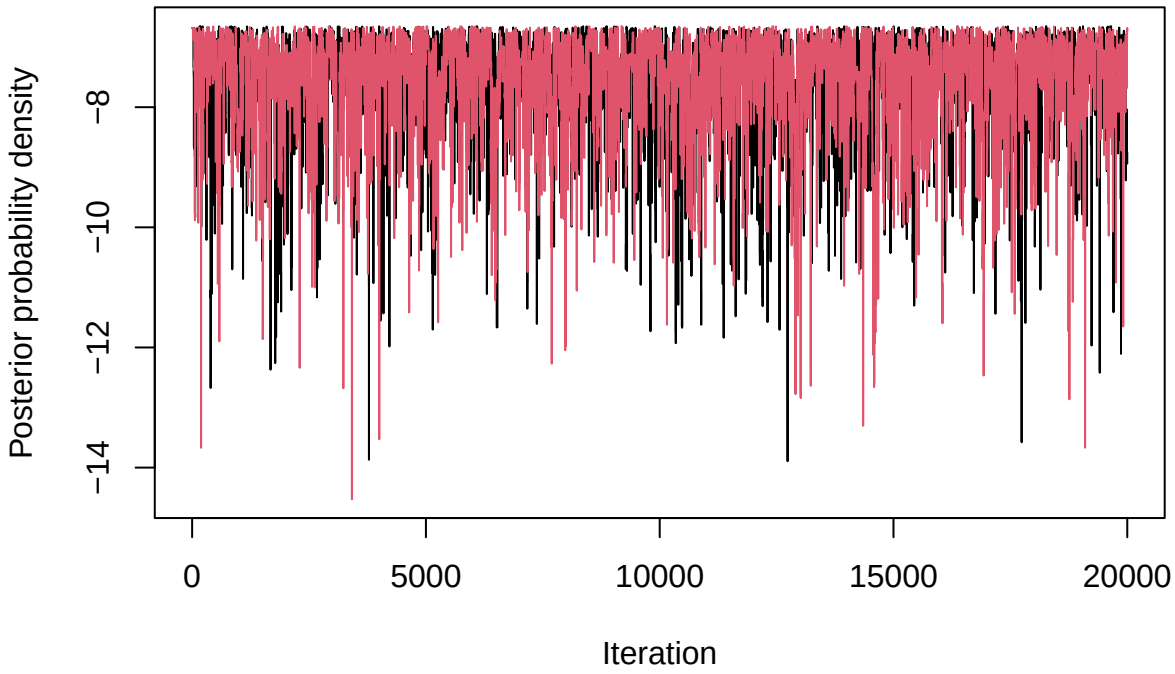
Step 5: Convergence diagnostics

We can check how well the chains mixed by plotting the posterior probability density over the samples.

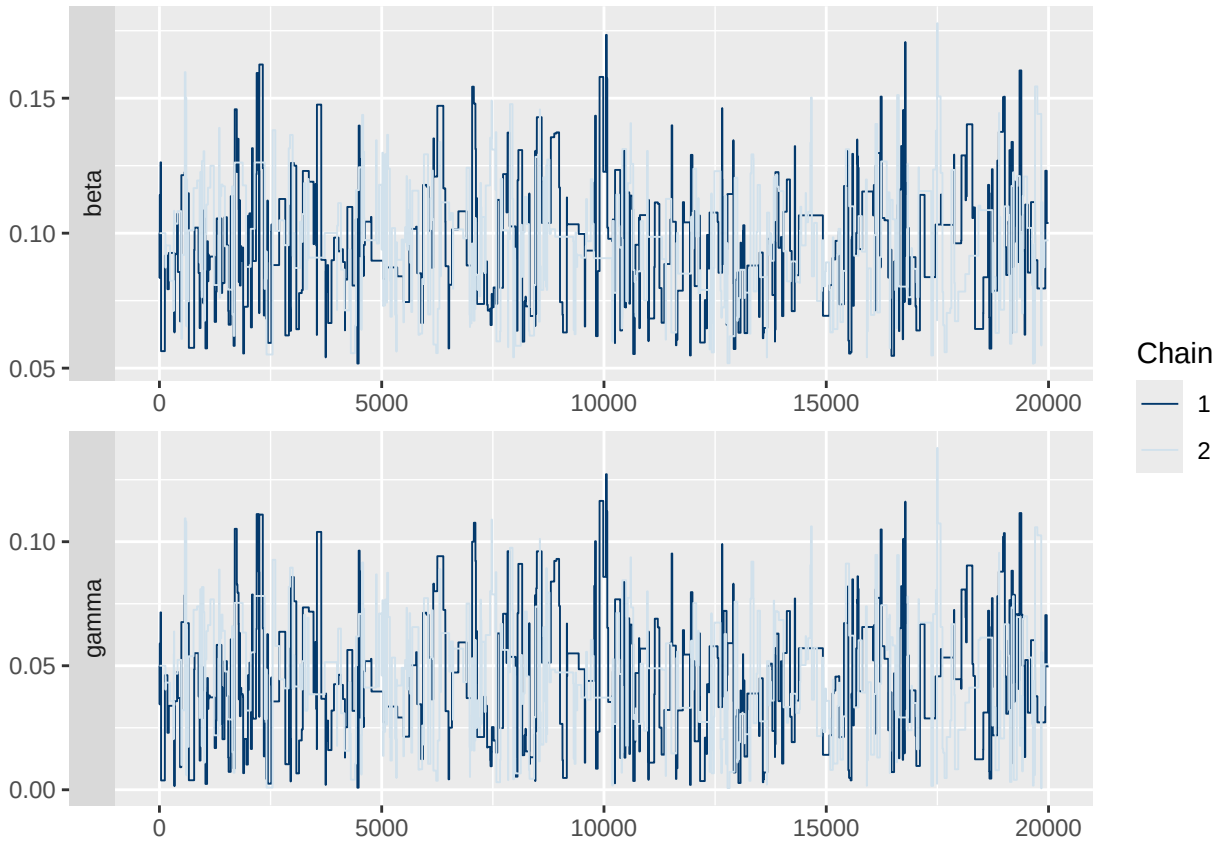
```
matplot(samples_rw$density, type = "l", lty = 1,
        xlab = "Iteration", ylab = "Posterior probability density")
```



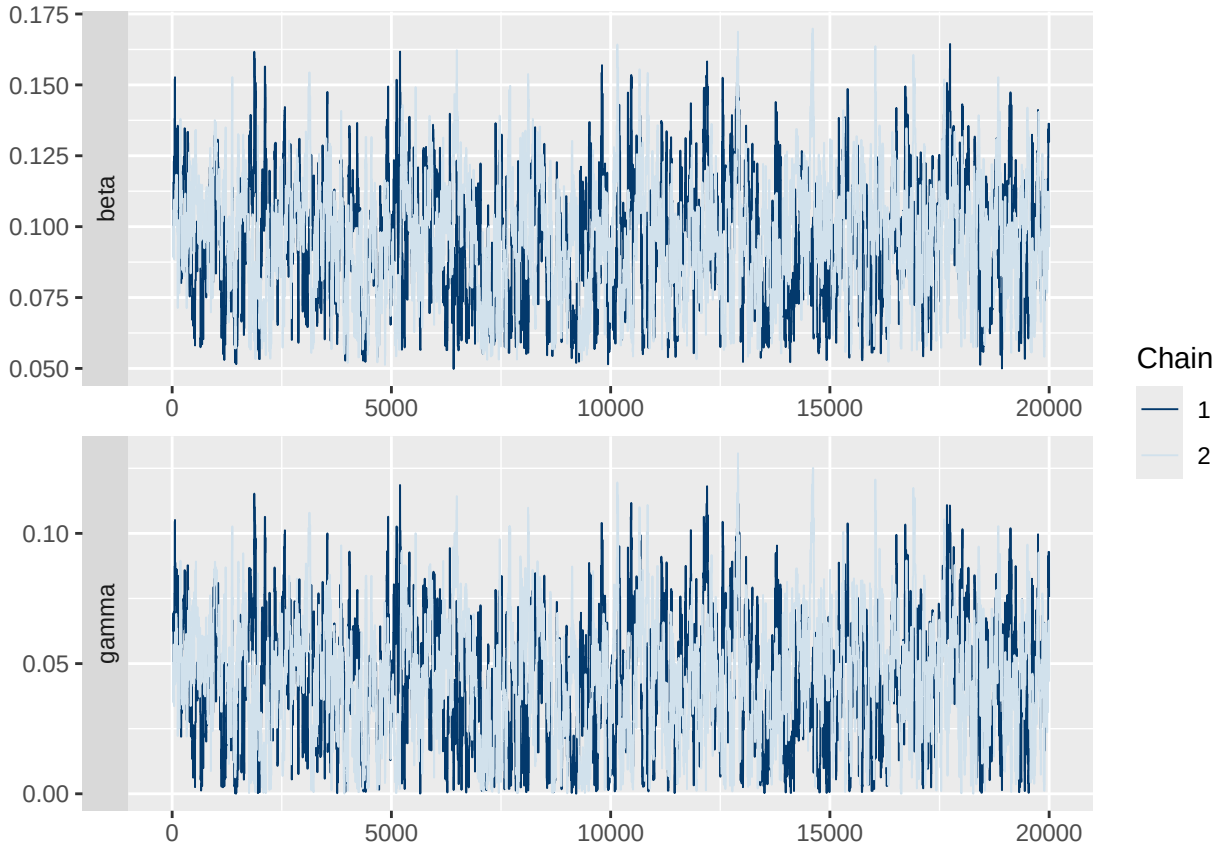

```
# Repeat this here for the MH algortihm
matplot(samples_mh$density, type = "l", lty = 1,
        xlab = "Iteration", ylab = "Posterior probability density")
```



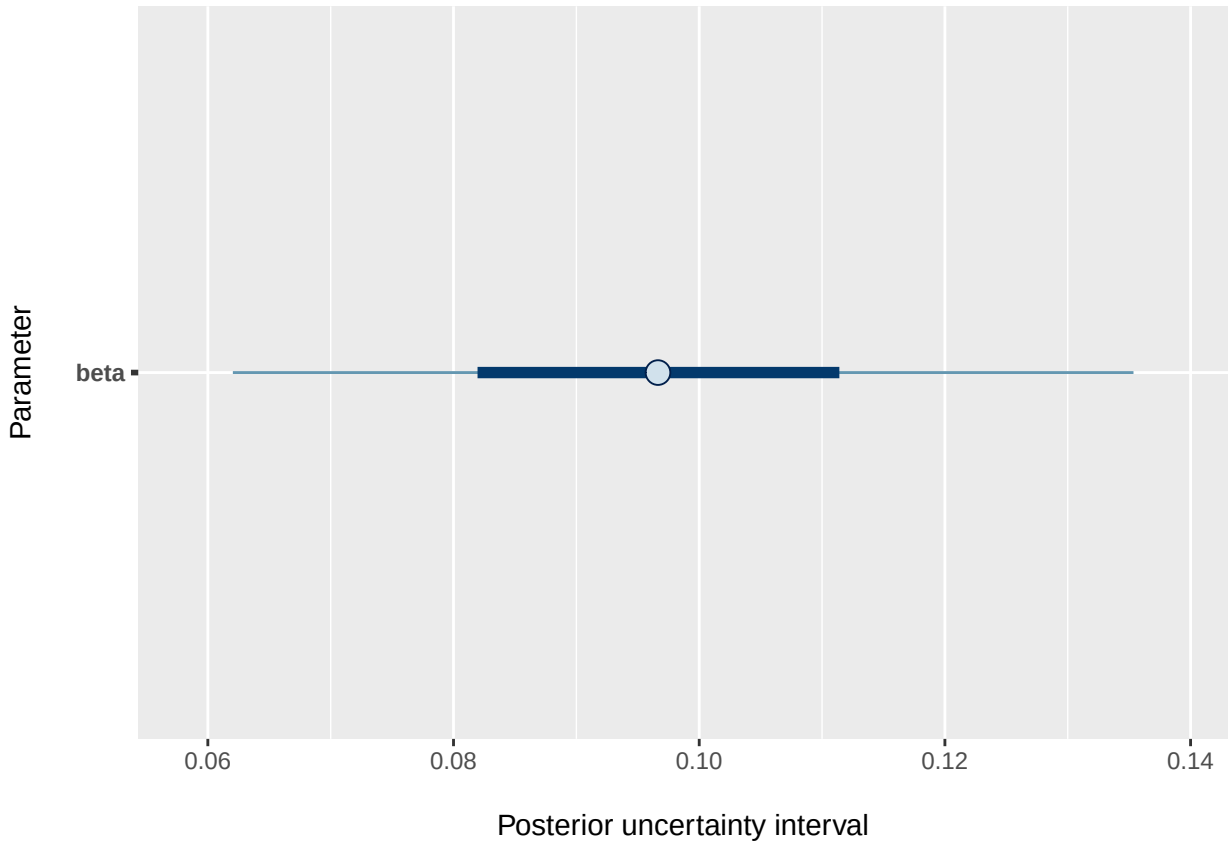

```
# We can now also check the trace for each parameter.
# It should look like a hairy caterpillar
mcmc_trace(draws_rw, facet_args = list(ncol = 1, strip.position = "left"))
```



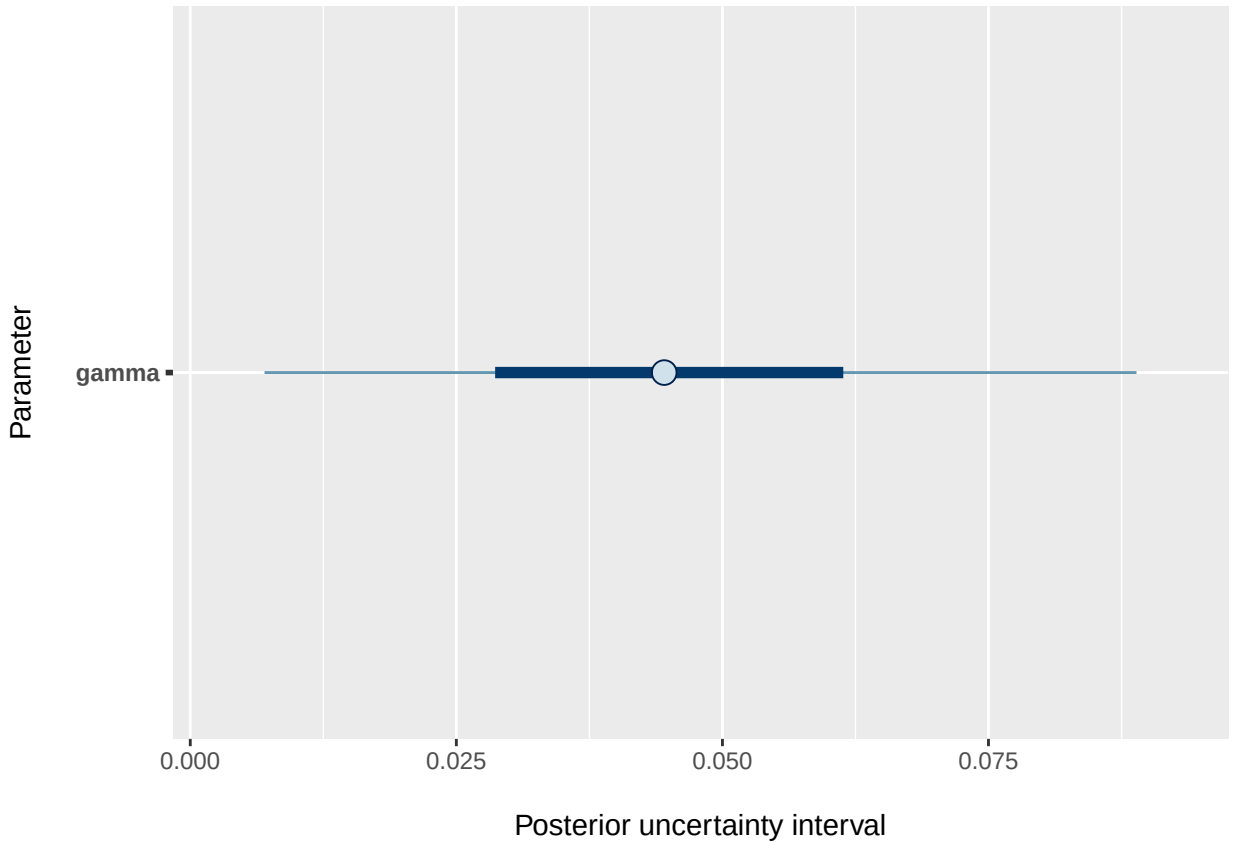

```
# Repeat this here for the MH algorithm
mcmc_trace(draws_mh, facet_args = list(ncol = 1, strip.position = "left"))
```



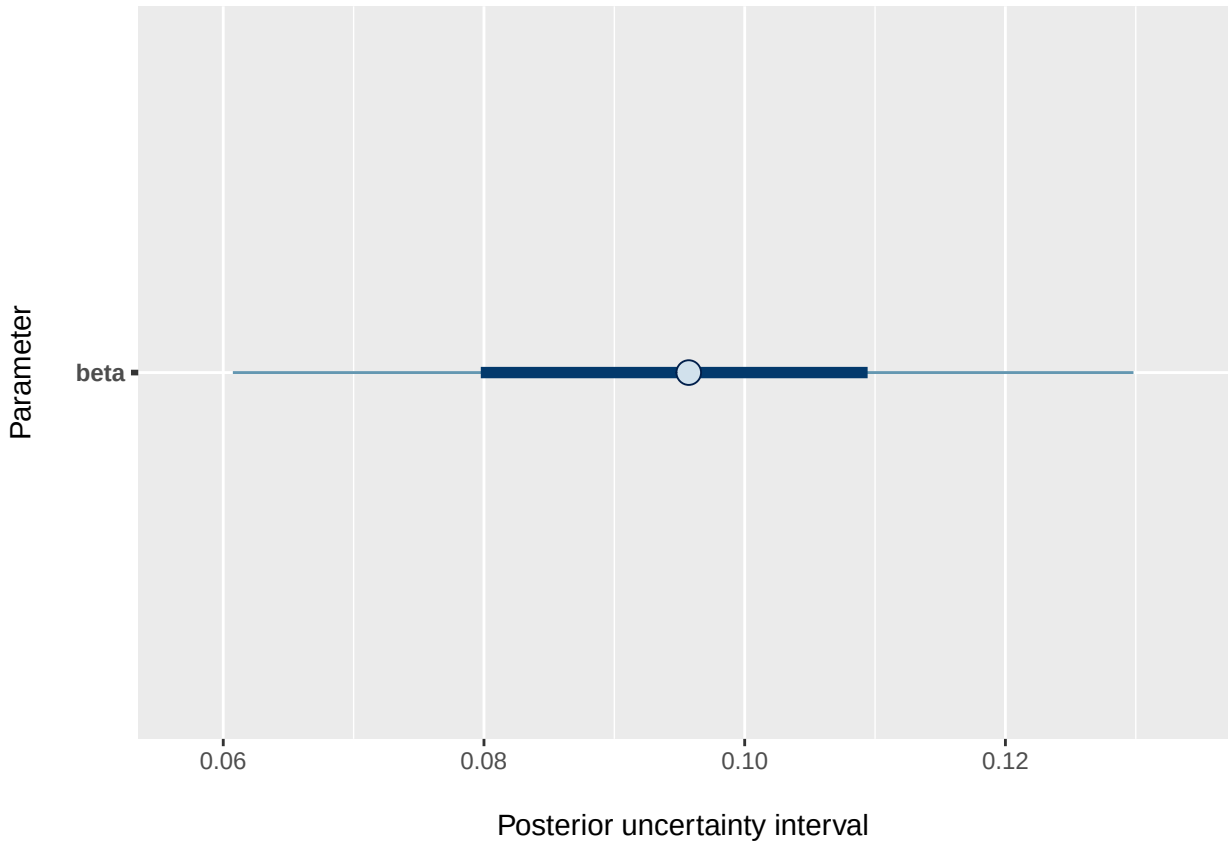

```
# We can plot posterior uncertainty intervals
draws_array_rw <- as_draws_array(draws_rw)
mcmc_intervals(draws_array_rw, pars = "beta") +
  labs(y = "Parameter\n", x = "\nPosterior uncertainty interval")
```



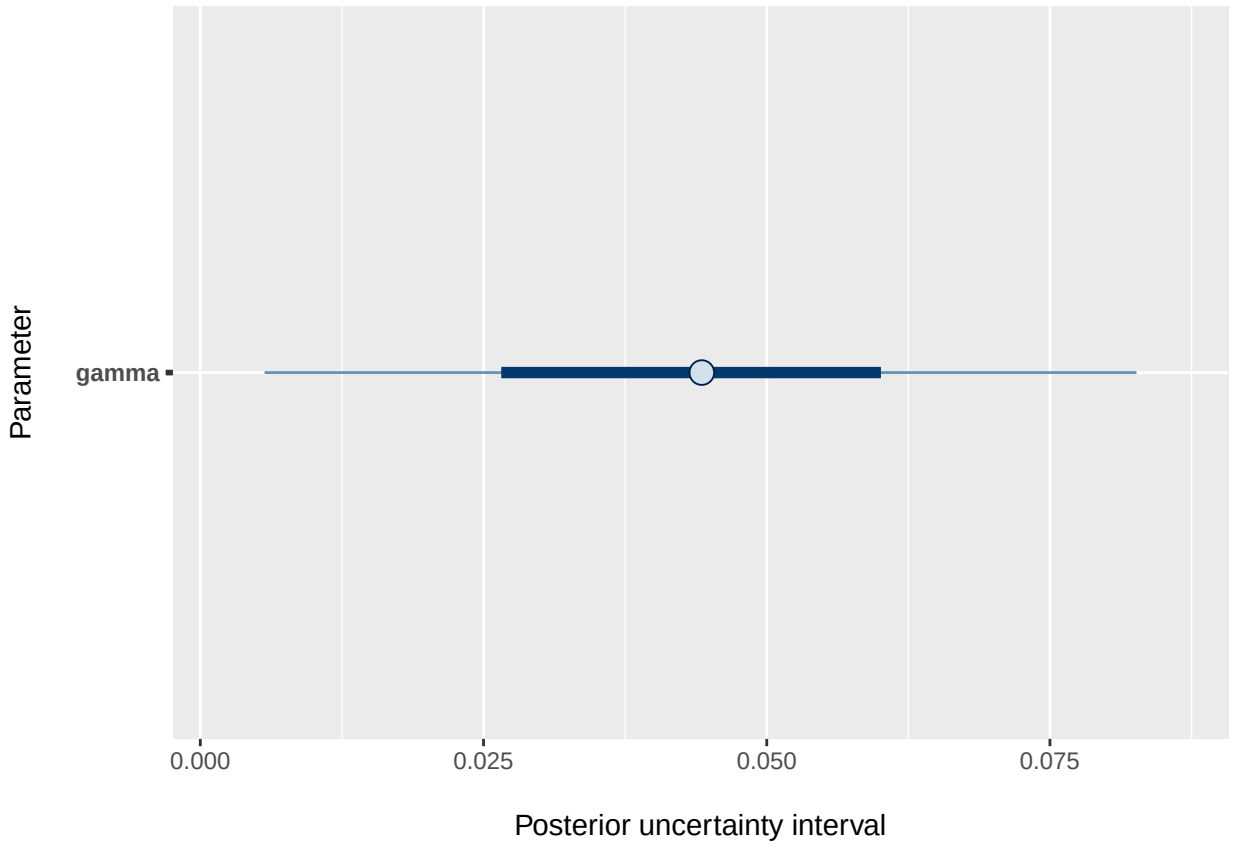

```
mcmc_intervals(draws_array_rw, pars = "gamma") +
  labs(y = "Parameter\n", x = "\nPosterior uncertainty interval")
```




```
# Repeat this here for the MH algortihm
draws_array_mh <- as_draws_array(draws_mh)
mcmc_intervals(draws_array_mh, pars = "beta") +
  labs(y = "Parameter\n", x = "\nPosterior uncertainty interval")
```



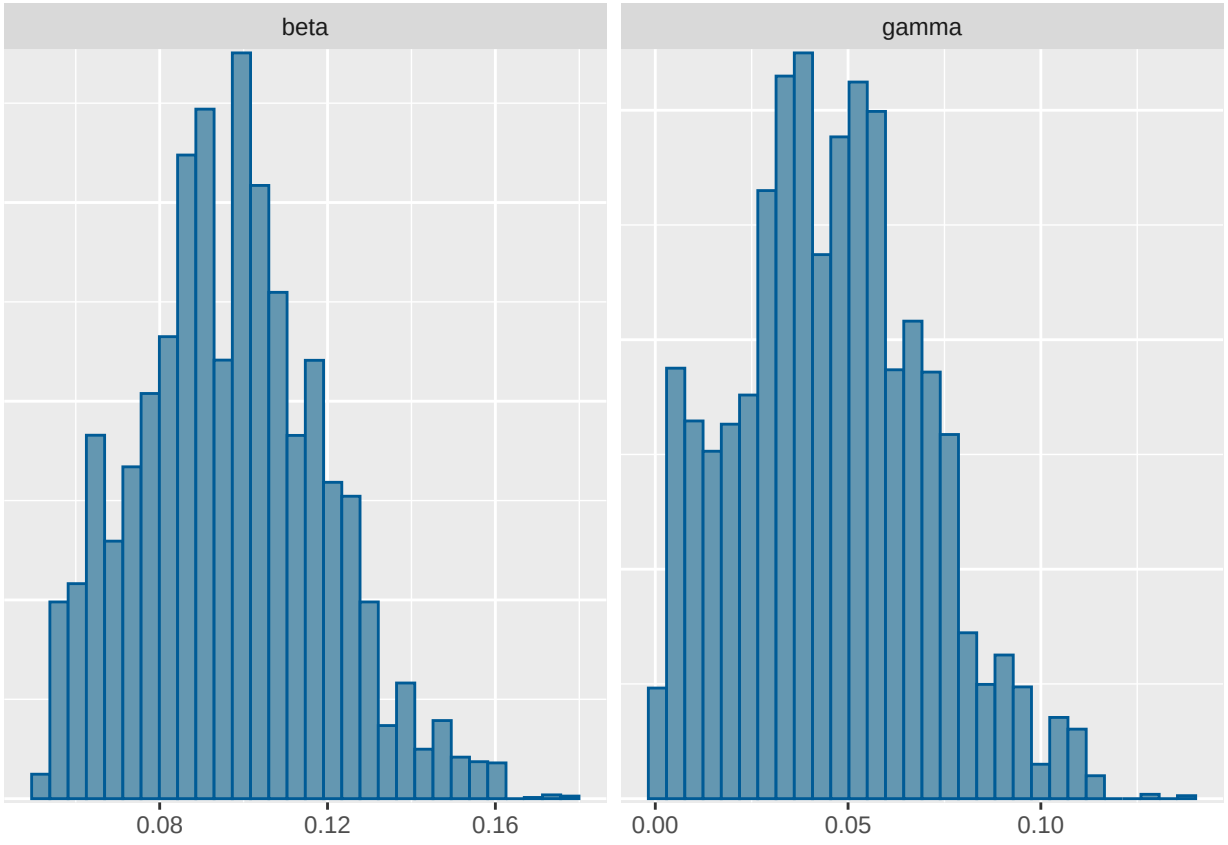

```
mcmc_intervals(draws_array_mh, pars = "gamma") +
  labs(y = "Parameter\n", x = "\nPosterior uncertainty interval")
```




```
# We can also plot univariate marginal posterior distributions
mcmc_hist(draws_array_rw)
```

```
## Warning in geom_histogram(set_hist_aes(freq), fill = get_color("mid"), color =
## get_color("mid_highlight"), : Ignoring unknown parameters: `size`
```

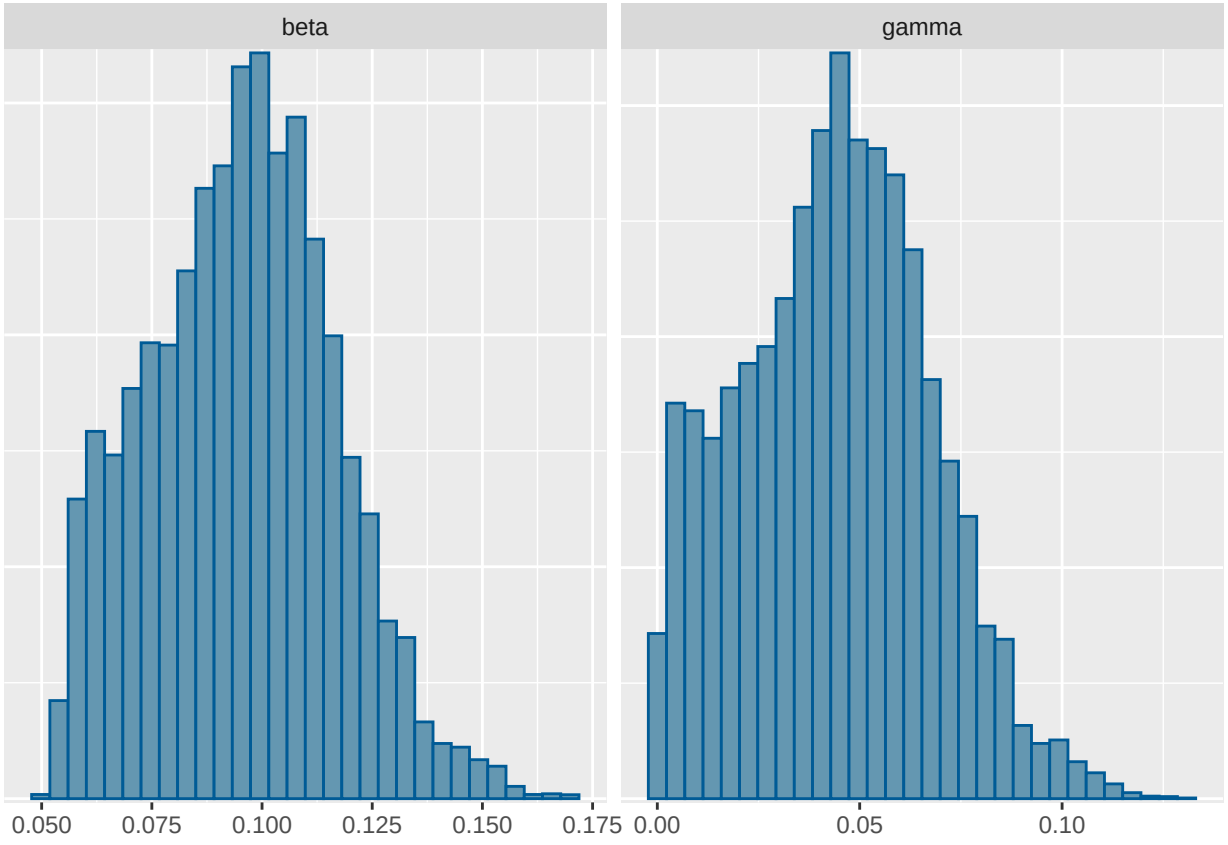
```
## `stat_bin()` using `bins = 30`. Pick better value `binwidth`.
```



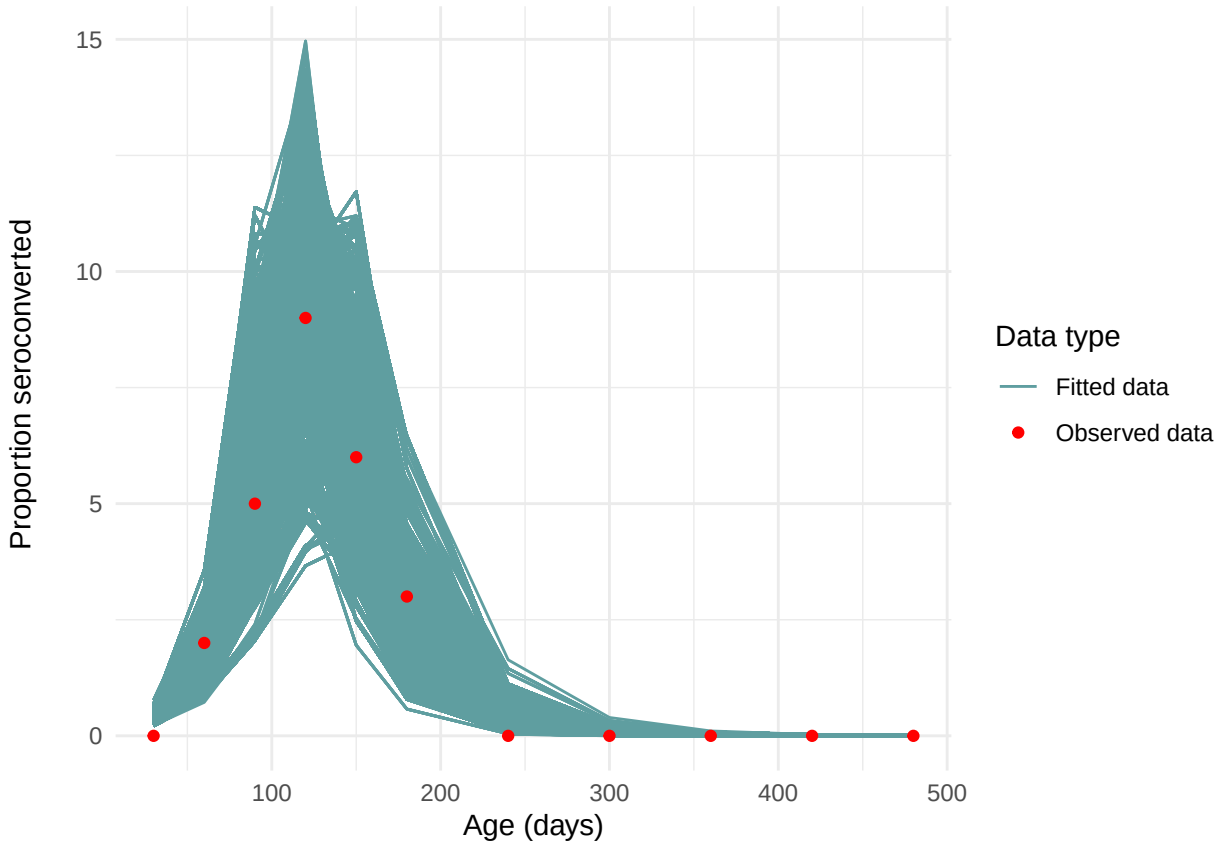

```
# Repeat this here for the MH algortihm
mcmc_hist(draws_array_mh)
```

```
## Warning in geom_histogram(set_hist_aes(freq), fill = get_color("mid"), color =
## get_color("mid_highlight"), : Ignoring unknown parameters: `size`
```

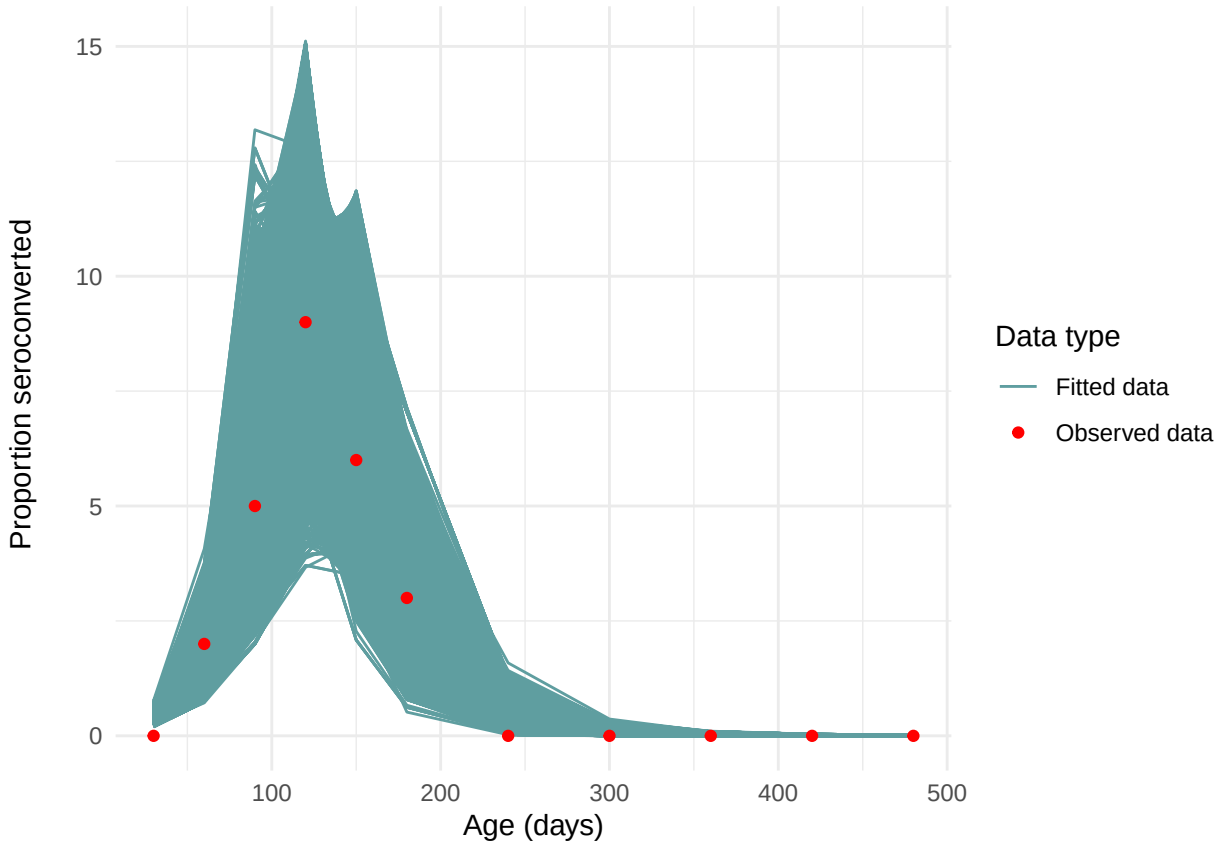
```
## `stat_bin()` using `bins = 30`. Pick better value `binwidth`.
```




```r
# Finally, we can plot the trajectories that we saved earlier
trajectories_rw <- dust_unpack_state(filter,
                                     samples_rw$observations$trajectories)
incidence_rw <- array(trajectories_rw$incidence, c(11, 20000))
# 11 time steps, 20000 samples

# Save this as a dataframe for easier plotting
incidence_rw <- data.frame(incidence$time, incidence_rw)
colnames(incidence_rw) <- c("time", paste0("sample_", 1:20000))

# Plot the fit
incidence_rw %>%
  pivot_longer(cols = -time, names_to = "sample",
               values_to = "seroconversion") %>%
  ggplot(aes(x = time, y = seroconversion)) +
  geom_line(aes(group = sample, col = "Fitted data")) +
  geom_point(data = incidence, aes(x = time, y = cases, col = "Observed data")) +
  labs(x = "Age (days)",
       y = " Proportion seroconverted\n",
       colour = "Data type") +
  scale_color_manual(values = c("Fitted data" = "#5F9EA0",
                                "Observed data" = "red")) +
  theme_minimal()
```




```r
# Repeat this here for the MH algorithm
trajectories_mh <- dust_unpack_state(filter,
                                     samples_mh$observations$trajectories)
incidence_mh <- array(trajectories_mh$incidence, c(11, 20000))
# 11 time steps, 20000 samples

# Save this as a dataframe for easier plotting
incidence_mh <- data.frame(incidence$time, incidence_mh)
colnames(incidence_mh) <- c("time", paste0("sample_", 1:20000))

# Plot the fit
incidence_mh %>%
  pivot_longer(cols = -time, names_to = "sample",
               values_to = "seroconversion") %>%
  ggplot(aes(x = time, y = seroconversion)) +
  geom_line(aes(group = sample, col = "Fitted data")) +
  geom_point(data = incidence, aes(x = time, y = cases, col = "Observed data")) +
  labs(x = "Age (days)",
       y = " Proportion seroconverted\n",
       colour = "Data type") +
  scale_color_manual(values = c("Fitted data" = "#5F9EA0",
                                "Observed data" = "red")) +
  theme_minimal()
```

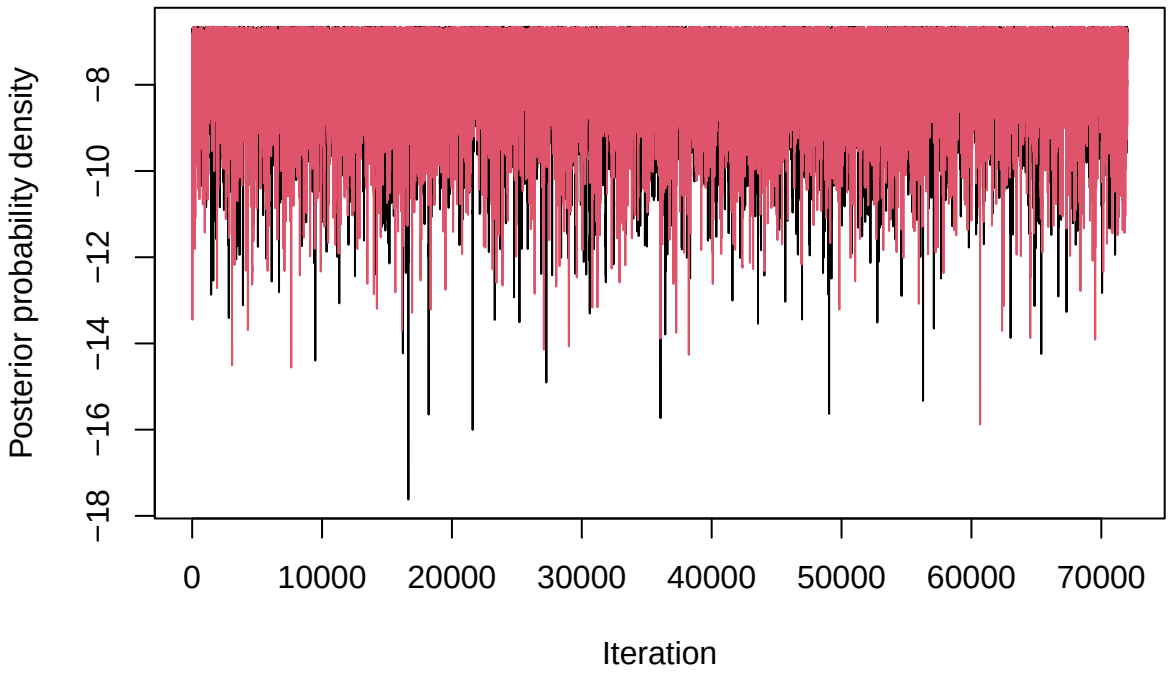


Step 6: Tuning the MCMC (optional)

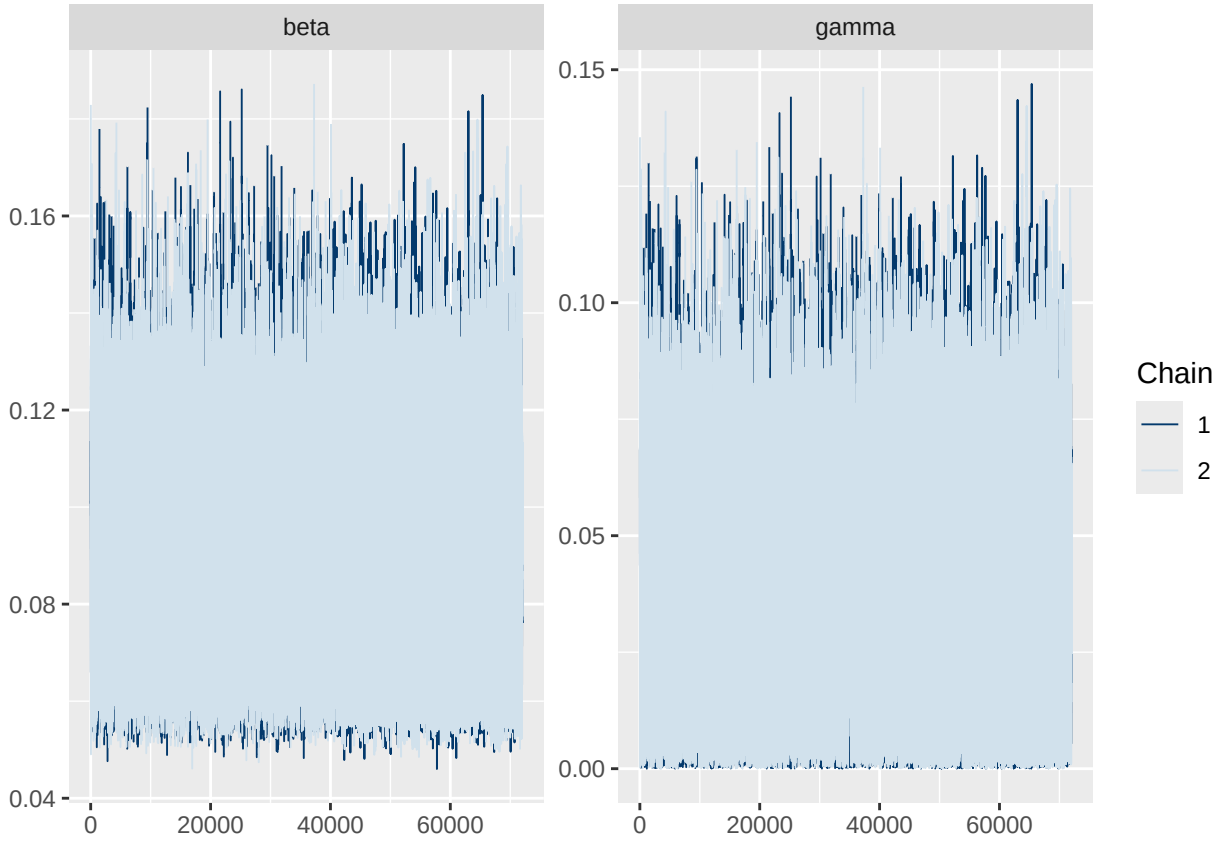
The results are pretty good but they could be improved by choosing more informative priors, better initial values, or tuning the samplers. We'll try tuning the sampler and focus on the random walk sampler but you can try other methods.

```
# Tune the sampler
# Define a new sampler (random walk)
vcv_tuned <- cov(draws_rw[1:2])
# The variance covariance matrix is now defined by the covariance of the
# previous samples
sampler_tuned <- monty_sampler_random_walk(vcv_tuned)

# Sample (this takes about 40 minutes on my laptop for the stochastic model and
# about 8 minutes for the deterministic one)
samples_tuned <- monty_sample(model,
                              sampler_tuned,
                              80000, # This time we run 8,000 iterations
                              initial = packer$pack(true_pars),
                              n_chains = 2,
                              runner = runner)
```

```
# Discard burn-in and check mixing again
samples_thinned <- monty_samples_thin(samples_tuned, burnin = 8000)

matplot(samples_thinned$density, type = "l", lty = 1,
        xlab = "Iteration", ylab = "Posterior probability density")
```



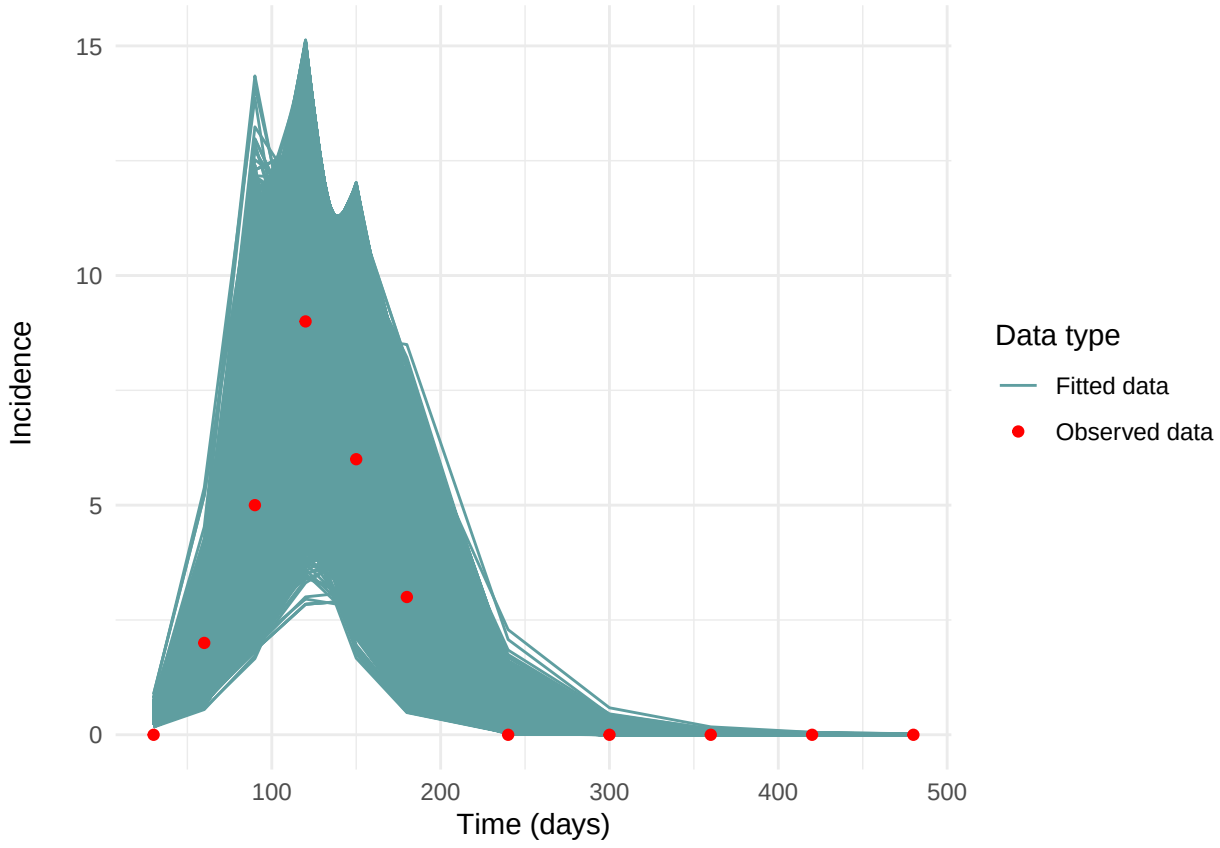

```
# Check the trace for each parameter
draws_thinned <- as_draws_df(samples_thinned)
mcmc_trace(draws_thinned)
```




```
# Look at the summary statistics
summarise_draws(draws_thinned)
```

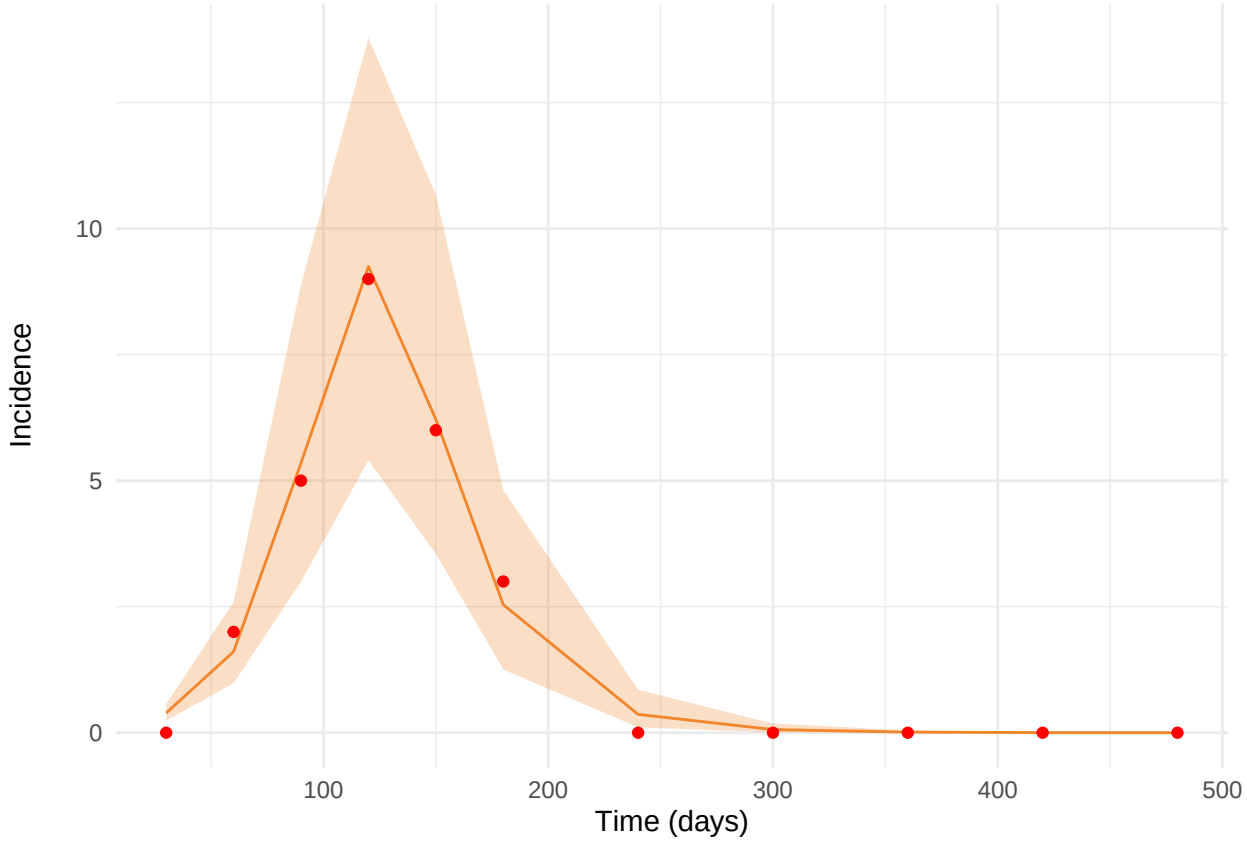
```
## # A tibble: 2 x 10
##   variable   mean median     sd    mad      q5    q95  rhat ess_bulk ess_tail
##   <chr>     <dbl>  <dbl>  <dbl>  <dbl>   <dbl>  <dbl> <dbl>    <dbl>    <dbl>
## 1 beta     0.0946 0.0943 0.0222 0.0231 0.0588  0.132   1.00   14879.   21027.
## 2 gamma    0.0433 0.0431 0.0245 0.0260 0.00395 0.0848  1.00   14958.   20960.
```

```r
# And plot the fit
# Trajectories
trajectories <- dust_unpack_state(filter,
                                  samples_thinned$observations$trajectories)
incidence_fit <- array(trajectories$incidence, c(11, 72000))
# Save this as a dataframe for easier plotting
incidence_df <- data.frame(incidence$time, incidence_fit )
colnames(incidence_df) <- c("time", paste0("sample_", 1:72000))

# Plot the fit
incidence_df %>%
  pivot_longer(cols = -time, names_to = "sample", values_to = "cases") %>%
  ggplot(aes(x = time, y = cases)) +
  geom_line(aes(group = sample, col = "Fitted data")) +
  geom_point(data = incidence, aes(x = time, y = cases, col = "Observed data")) +
  labs(x = "Time (days)",
       y = " Incidence\n",
       colour = "Data type") +
  scale_color_manual(values = c("Fitted data" = "#5F9EA0",
                                "Observed data" = "red")) +
  theme_minimal()
```



A better way of displaying the fit is to plot the credible intervals of the posterior predictive distribution.

```r
# Calculate credible intervals
incidence_summary <- incidence_df %>%
  pivot_longer(cols = -time, names_to = "sample", values_to = "cases") %>%
  group_by(time) %>%
  summarise(median = median(cases),
            lower_95 = quantile(cases, 0.025),
            upper_95 = quantile(cases, 0.975))

# Plot credible intervals
incidence_summary %>%
  ggplot(aes(x = time)) +
  geom_ribbon(aes(ymin = lower_95, ymax = upper_95), fill = "#F2862C44") +
  geom_line(aes(y = median), color = "#F2862C") +
  geom_point(data = incidence, aes(x = time, y = cases), color = "red") +
  labs(x = "Time (days)",
       y = "Incidence\n") +
  theme_minimal()
```



Step 7: Compare the estimates to the true values

Finally, we can compare the estimates to the true values.

```r
true_values <- data.frame(Parameter = c("beta", "gamma"),
                          True_Value = c(true_pars$beta,
                                         true_pars$gamma))
```

```
estimates <- data.frame(Parameter = c("beta", "gamma"),
                        Estimate = c(round(median(draws_thinned$beta), 3),
                                     round(median(draws_thinned$gamma), 3)))
comparison <- merge(true_values, estimates, by = "Parameter")
kable(comparison, caption = "Comparison of true values and estimates",
      align = "cc") |>
  column_spec(1:3, width = "30mm")
```

Table 3: Comparison of true values and estimates

Parameter	True_Value	Estimate
beta	0.10	0.094
gamma	0.05	0.043