Simulation-based calibration of capture-mark-recapture models in Stan

This document contains the results of simulation-based calibration (SBC) for each model described in the manuscript. Calibration is assessed through ECDF plots and parameter estimates are visually inspected alongside their true values. The simulation script is printed at the end of this document.

Table of contents

Introduction **3**

Cormack-Jolly-Seber **5**
Single survey 5
Robust design 6

Multistate Cormack-Jolly-Seber **7**
Single survey 7
Robust design 10

Multievent Cormack-Jolly-Seber **13**
Single survey 13
Robust design 16

Jolly-Seber **19**
Single survey 19
Robust design 23

Multistate Jolly-Seber **26**
Single survey 26
Robust design 31

Multievent Jolly-Seber **36**
Single survey 36

Robust design 41

Simulation script **46**

Utility functions **52**

References **54**

Introduction

Simulation-based calibration (SBC) was used to assess whether Stan programs correctly encoded the data-generating processes (DGPs). Briefly, SBC works by simulating prior predictive datasets, sampling from the joint posterior distribution, and then assessing the ranks of the simulated values within the posterior draws of model parameters (Modrák et al. 2023). If the model reflects the DGP, then the ranks are uniformly distributed which is assessed using the `SBC::plot_ecdf_diff()` function, where the black squiggly lines should fall within the blue ellipses. Parameter estimates were also plotted alongside their input to visually assess parameter recovery. Models were fit with CmdStanR 0.9 (Gabry et al. 2025) using CmdStan 2.38 with 500 simulated datasets per model. In Jolly-Seber models, the forward-backward sampling algorithm was used to recover latent discrete parameters (Zucchini et al. 2017), which are used to produce population size and number of entries and exits per survey/primary, as well as the super-population.

Stan programs accommodate two model versions utilising different likelihood functions:

1. Functions accommodating survey-varying parameters which share likelihood terms between individuals, with only one likelihood computation for augmented individuals in Jolly-Seber models. These are the default models in this document but can be changed by setting `ind = FALSE` in the data provided to Stan or via the simulation script.
2. Functions accommodating individual-by-survey varying parameters, where most likelihood terms must be computed for each individual separately, including for all augmented individuals in Jolly-Seber models. These models are fit by setting `ind = TRUE` and `collapse = FALSE`. Note that because no individual-level predictors are observed for augmented individuals, these parameters must be imputed; the same would be true for individual-by-survey varying predictors for all surveys where individuals were not detected.

Additionally, for Jolly-Seber models, there is an additional "collapsed" version of (2) with only one likelihood computation for augmented individuals like in version (1). This version permits individual parameters for the observed sample of individuals, while computational burden of the augmented computations is reduced. These models can be fit by setting `ind = TRUE` and `collapse = TRUE`, and use the `js*2()` and `js*2_rng()` function signatures. For Cormack-Jolly-Seber models, the `grainsize` argument can be set to enable within-chain parallelisation.

Parameters were simulated as follows:

- Mortality hazard rates (h) as constant and state-specific in multistate/multievent;
- Detection probabilities (p) varying by survey/primary (not secondary in robust design) and state-specific in multistate/multievent;
- Transition rates (q) as constant and unique for each transition;
- Event matrices ($\boldsymbol{E}$) as lower bidiagional stochastic (rows sum to 1);

- Initial state probabilities (η) as constant (simplex);
- Entry probabilities ($\boldsymbol{\beta}$) as varying by survey/primary, with survey intervals as offsets, as logistic-normal (but can be fit by setting `dirichlet = TRUE`).

```
# load packages and functions
library(SBC)
library(cmdstanr)
library(here)
library(tidyverse)
source(here("sbc/util.R"))
source(here("sbc/dgp-cmr.R"))
theme_set(my_theme(base_size = 6))

# set simulation and HMC parameters
N_super <- 200 ; J <- 6 ; Jm1 <- J - 1 ; K_max <- 3 ; S <- 2 ; n_sims <- 500
chains <- 8 ; iter_warmup <- 200 ; iter_sampling <- 500 ; max_treedepth <- 10
options(digits = 3, mc.cores = chains)
```

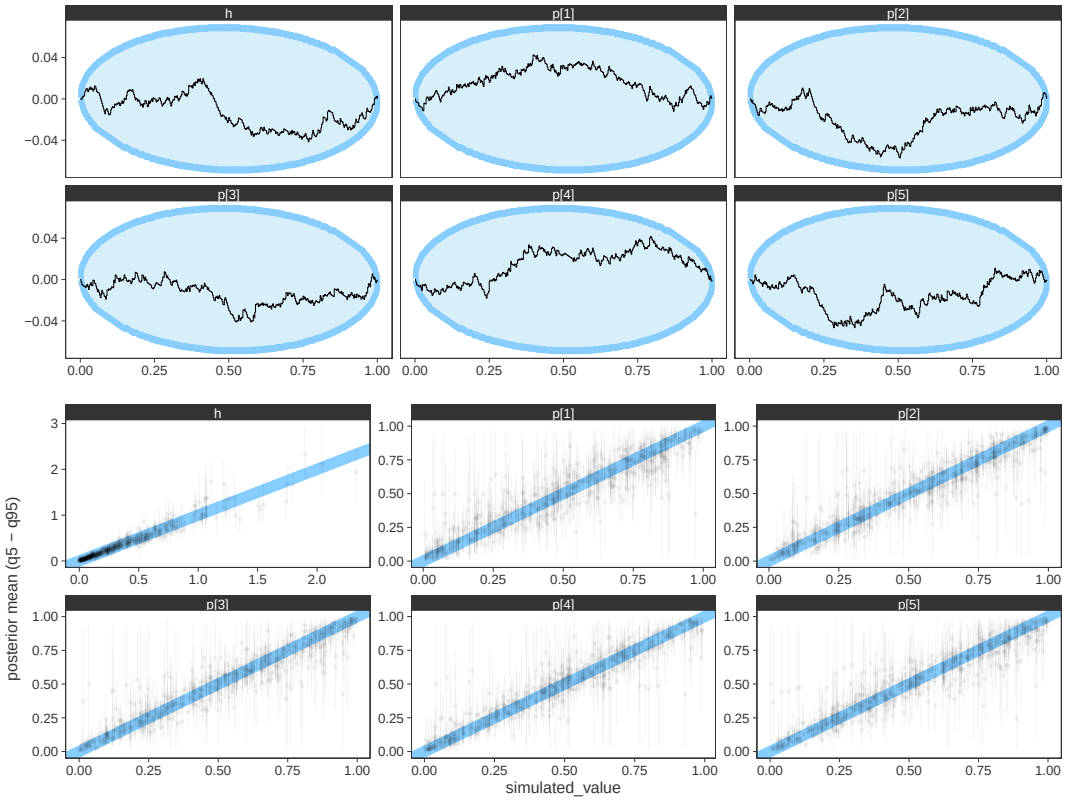
Cormack-Jolly-Seber

Single survey

```
datasets <- SBC_generator_function(simulate_cmr, N_super = N_super, J = J) |>
  generate_datasets(n_sims)
backend <- SBC_backend_cmdstan_sample(cmdstan_model(here("stan/cjs.stan")),
                                      init = 0.1, chains = chains,
                                      iter_warmup = iter_warmup,
                                      iter_sampling = iter_sampling,
                                      max_treedepth = max_treedepth)
sbc <- compute_SBC(datasets, backend, cores_per_fit = chains, keep_fits = F)
```

Mean time for the slowest of 8 chains with 700 iterations: 0.004 minutes.

```
plot_sbc(sbc, nrow = 2)
```

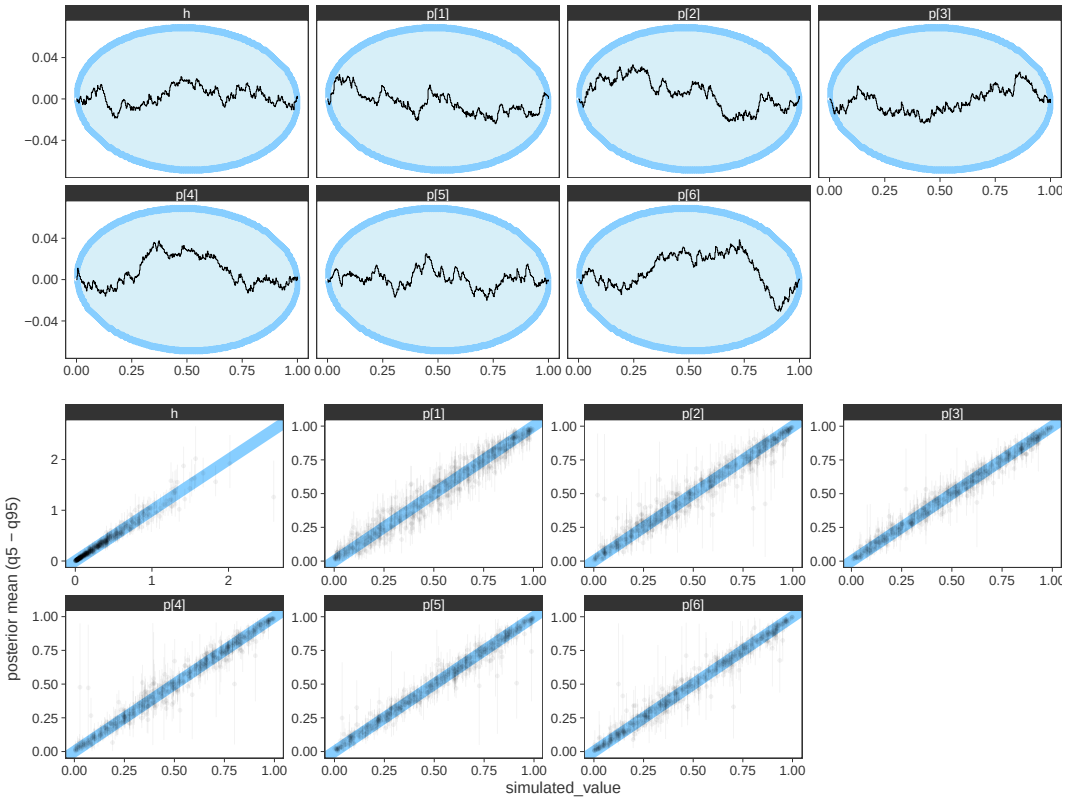


Robust design

```
datasets <- SBC_generator_function(simulate_cmr, N_super = N_super, J = J,
                                   K_max = K_max) |>
  generate_datasets(n_sims)
backend <- SBC_backend_cmdstan_sample(cmdstan_model(here("stan/cjs-rd.stan")),
                                      init = 0.1, chains = chains,
                                      iter_warmup = iter_warmup,
                                      iter_sampling = iter_sampling,
                                      max_treedepth = max_treedepth)
sbc <- compute_SBC(datasets, backend, cores_per_fit = chains, keep_fits = F)
```

Mean time for the slowest of 8 chains with 700 iterations: 0.006 minutes.

```
plot_sbc(sbc, nrow = 2)
```



Multistate Cormack-Jolly-Seber

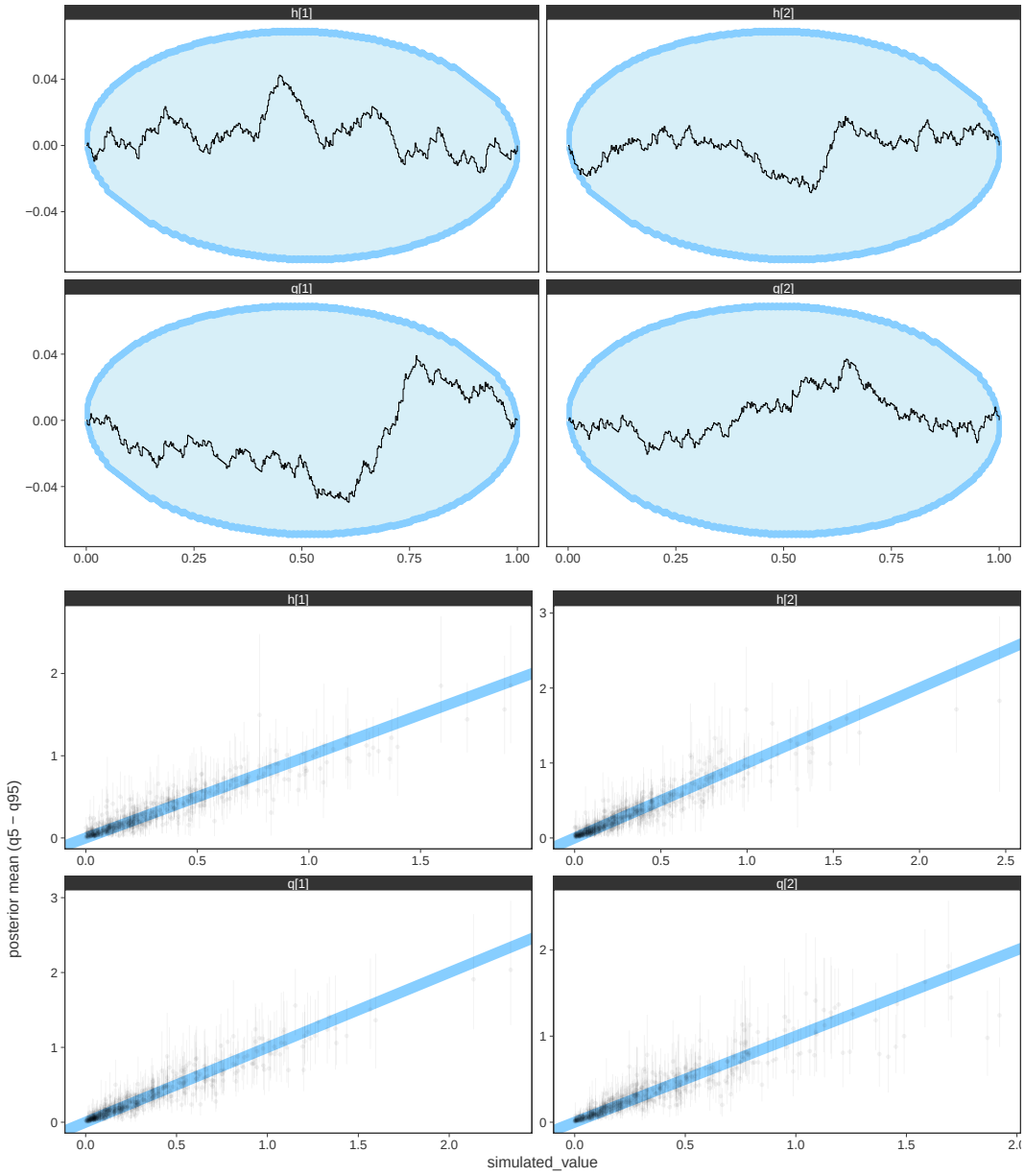
Single survey

```
datasets <- SBC_generator_function(simulate_cmr, N_super = N_super, J = J,
                                   S = S) |>
  generate_datasets(n_sims)
backend <- SBC_backend_cmdstan_sample(cmdstan_model(here("stan/cjs-ms.stan")),
                                      init = 0.1, chains = chains,
                                      iter_warmup = iter_warmup,
                                      iter_sampling = iter_sampling,
                                      max_treedepth = max_treedepth)
sbc <- compute_SBC(datasets, backend, cores_per_fit = chains, keep_fits = F)
```

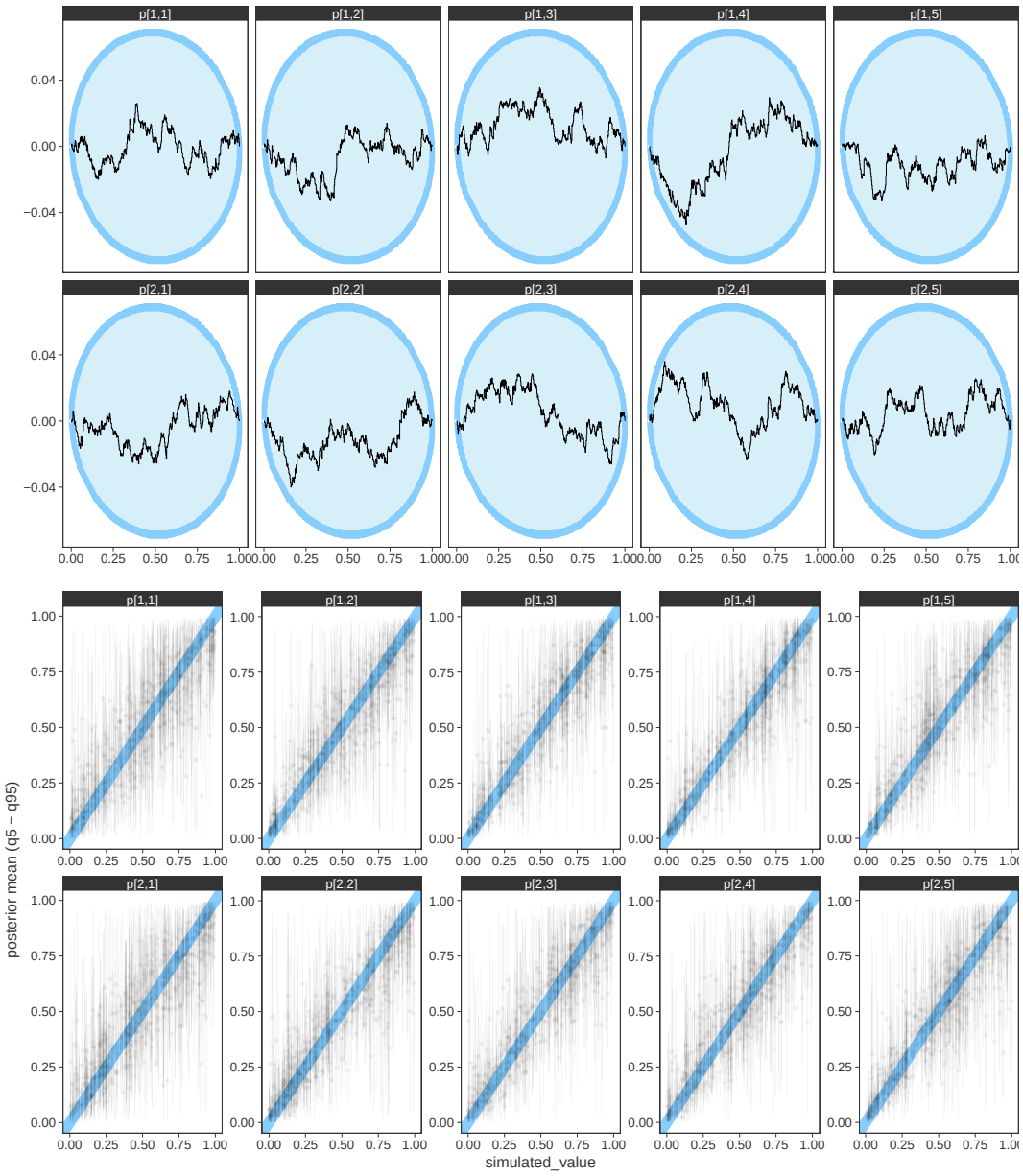
Mean time for the slowest of 8 chains with 700 iterations: 0.018 minutes.

> **Warning**
>
> In single survey multistate/multievent models, transition rates and state-specific detection probabilities are confounded (Hollanders and Royle 2022). Notice the difference in parameter estimates with the robust design.

```
plot_sbc(sbc, c(str_c("h[", 1:S, "]"), str_c("q[", 1:(S * (S - 1)), "]")))
```



```
plot_sbc(sbc, str_c("p[", rep(1:S, each = Jm1), ",", rep(1:Jm1, S), "]"),
         ncol = Jm1)
```

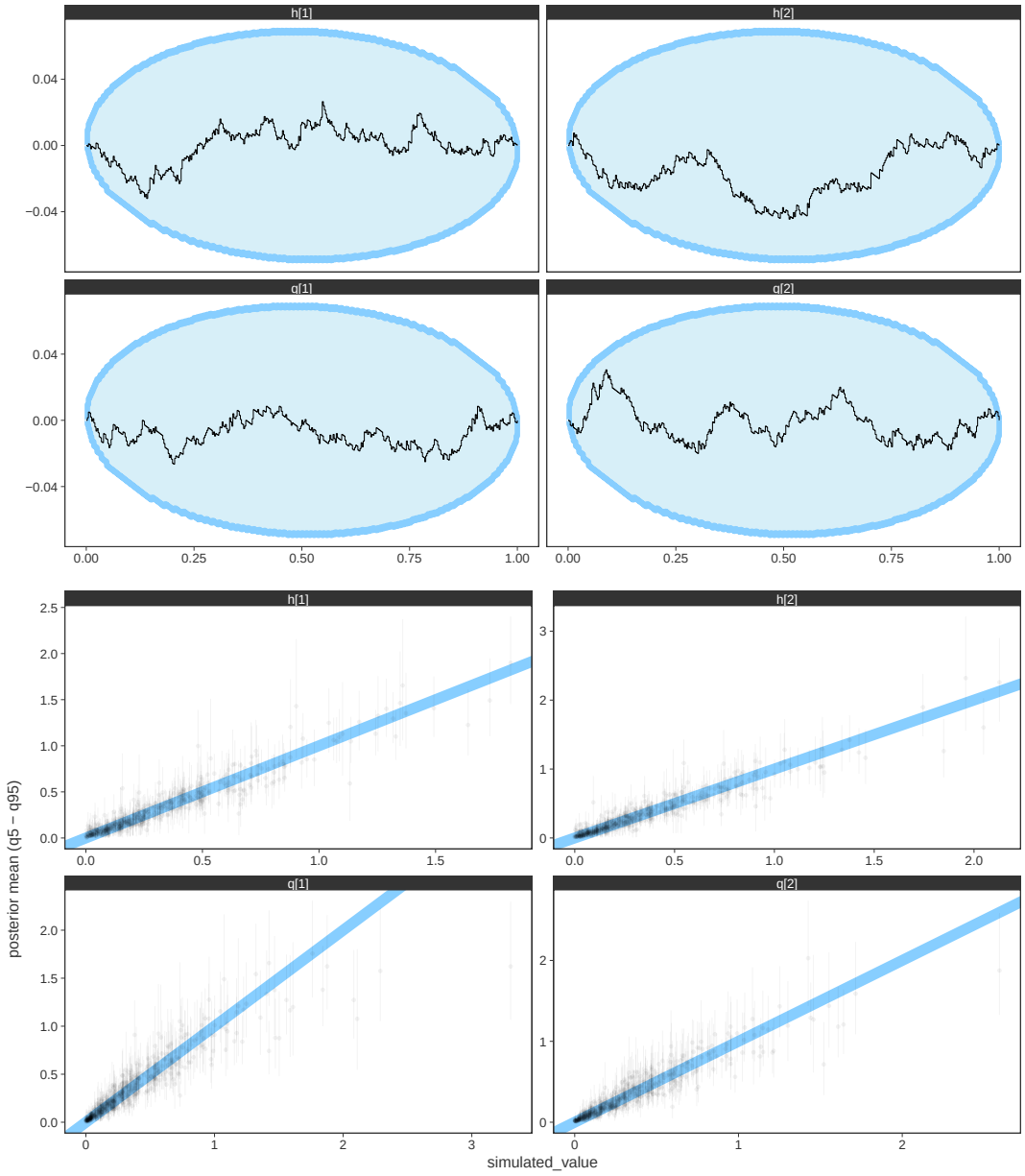


Robust design

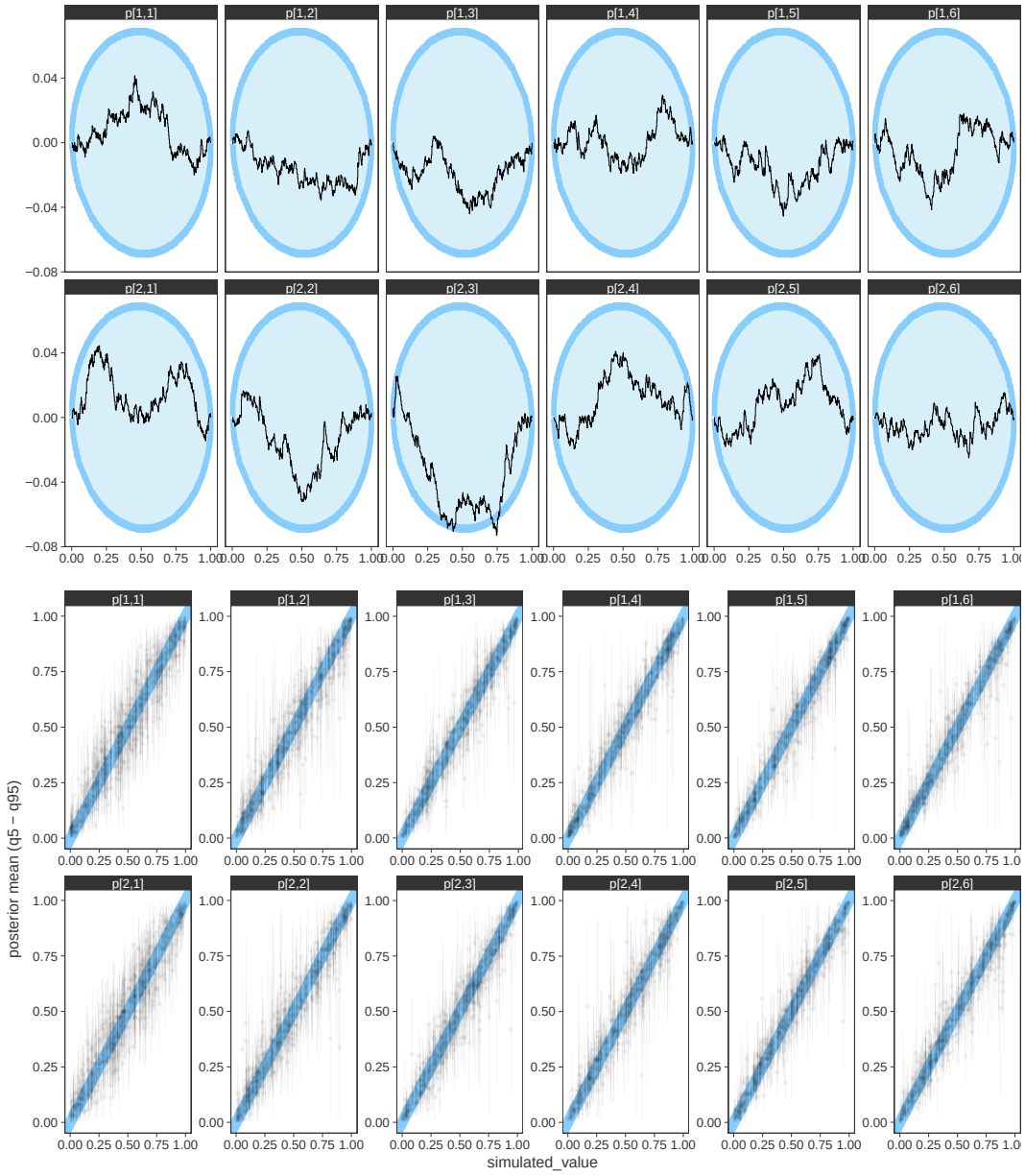
```
datasets <- SBC_generator_function(simulate_cmr, N_super = N_super, J = J,
                                   K_max = K_max, S = S) |>
  generate_datasets(n_sims)
backend <- SBC_backend_cmdstan_sample(cmdstan_model(here("stan/cjs-ms-rd.stan")),
                                      init = 0.1, chains = chains,
                                      iter_warmup = iter_warmup,
                                      iter_sampling = iter_sampling,
                                      max_treedepth = max_treedepth)
sbc <- compute_SBC(datasets, backend, cores_per_fit = chains, keep_fits = F)
```

Mean time for the slowest of 8 chains with 700 iterations: 0.022 minutes.

```
plot_sbc(sbc, c(str_c("h[", 1:S, "]"), str_c("q[", 1:(S * (S - 1)), "]")))
```



```
plot_sbc(sbc, str_c("p[", rep(1:S, each = J), ",", rep(1:J, S), "]"), ncol = J)
```



Multievent Cormack-Jolly-Seber

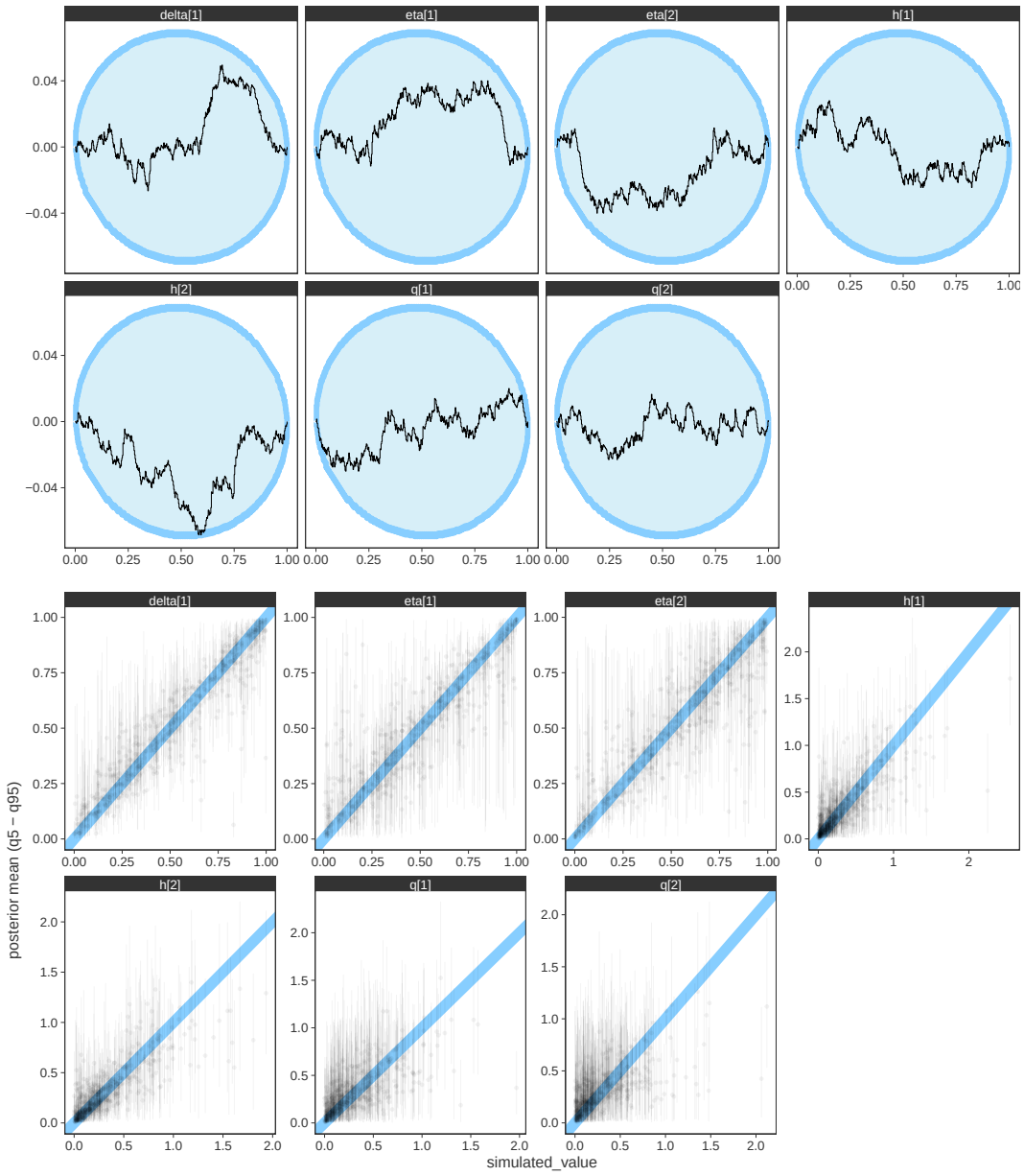
Lower bidiagonal stochastic matrices were specified for the event matrix $\boldsymbol{E}$, with state-specific assignment probabilities $\boldsymbol{\delta} = (\delta_1, \ldots, \delta_{S-1})$ corresponding to assigning to states $s \forall \{2, \ldots, S\}$, with state $s-1$ having state misclassification probability $1 - \delta_s$.

Single survey

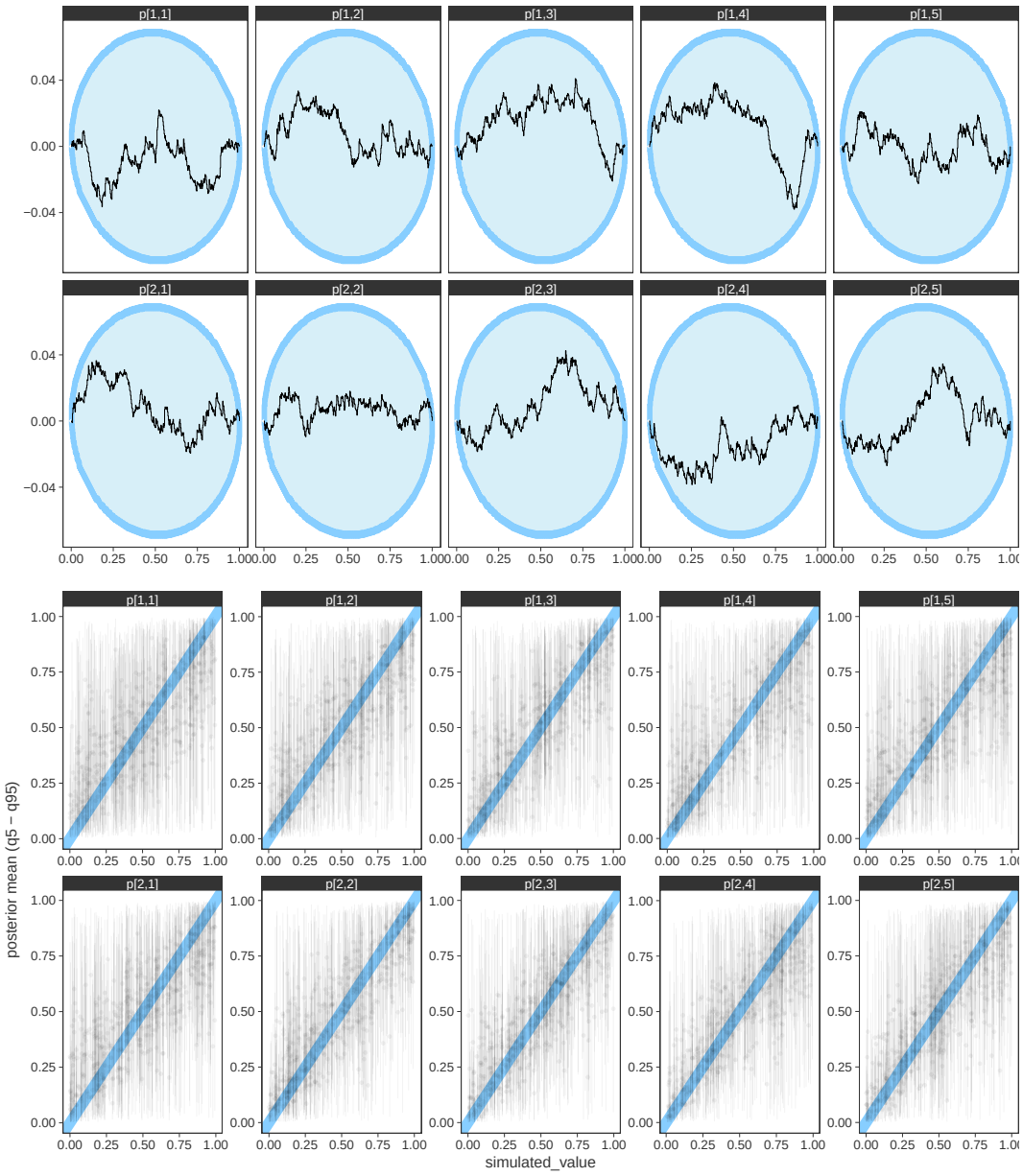
```
datasets <- SBC_generator_function(simulate_cmr, N_super = N_super, J = J,
                                   S = S, ME = T) |>
  generate_datasets(n_sims)
backend <- SBC_backend_cmdstan_sample(cmdstan_model(here("stan/cjs-me.stan")),
                                      init = 0.1, chains = chains,
                                      iter_warmup = iter_warmup,
                                      iter_sampling = iter_sampling,
                                      max_treedepth = max_treedepth)
sbc <- compute_SBC(datasets, backend, cores_per_fit = chains, keep_fits = F)
```

Mean time for the slowest of 8 chains with 700 iterations: 0.145 minutes.

```
plot_sbc(sbc, c(str_c(rep(c("h", "eta"), each = S), "[", rep(1:S, 2), "]"),
                str_c("q[", 1:(S * (S - 1)), "]"),
                str_c("delta[", 1:(S - 1), "]")), nrow = 2)
```



```
plot_sbc(sbc, str_c("p[", rep(1:S, each = Jm1), ",", rep(1:Jm1, S), "]"),
         ncol = Jm1)
```

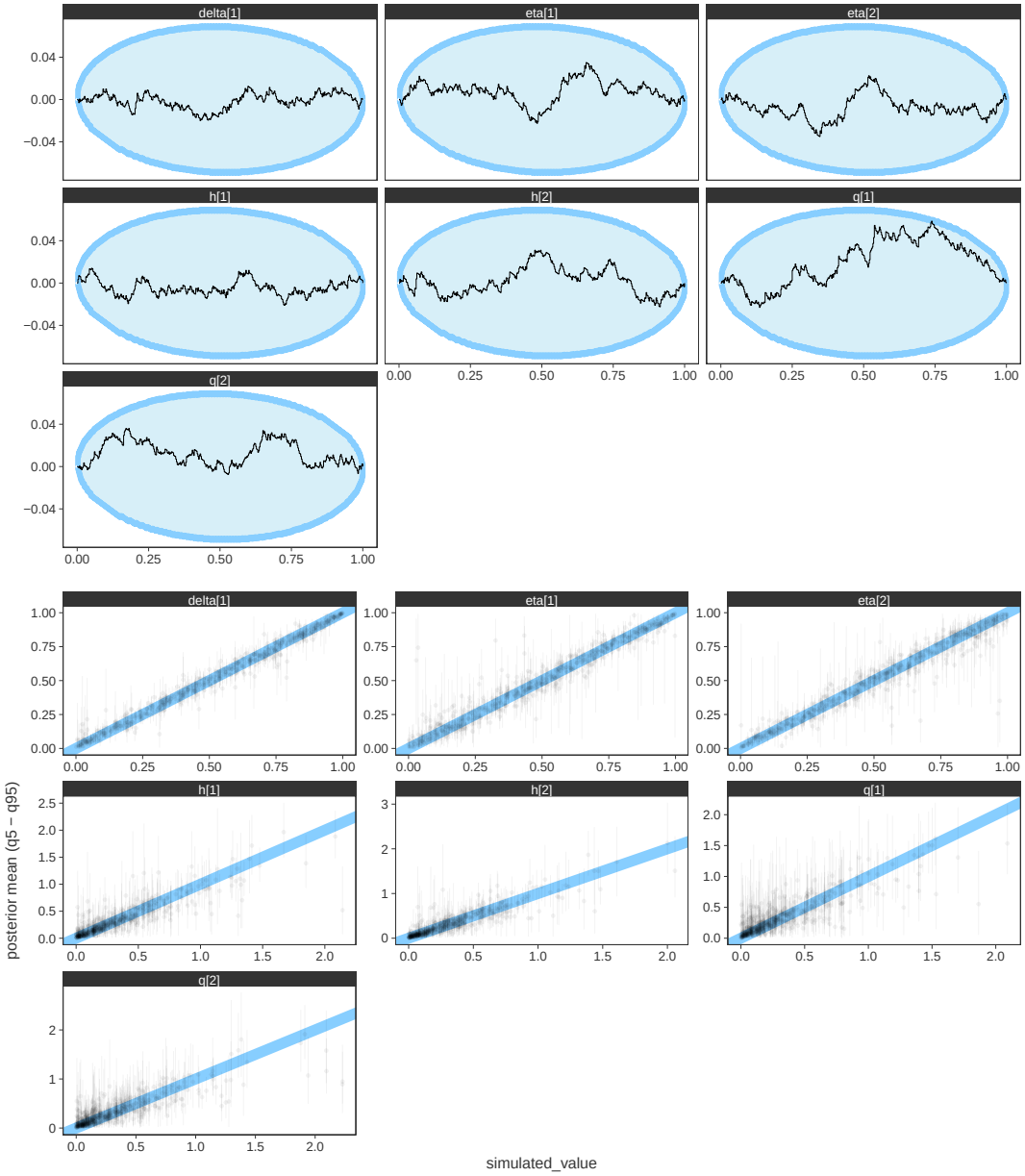


Robust design

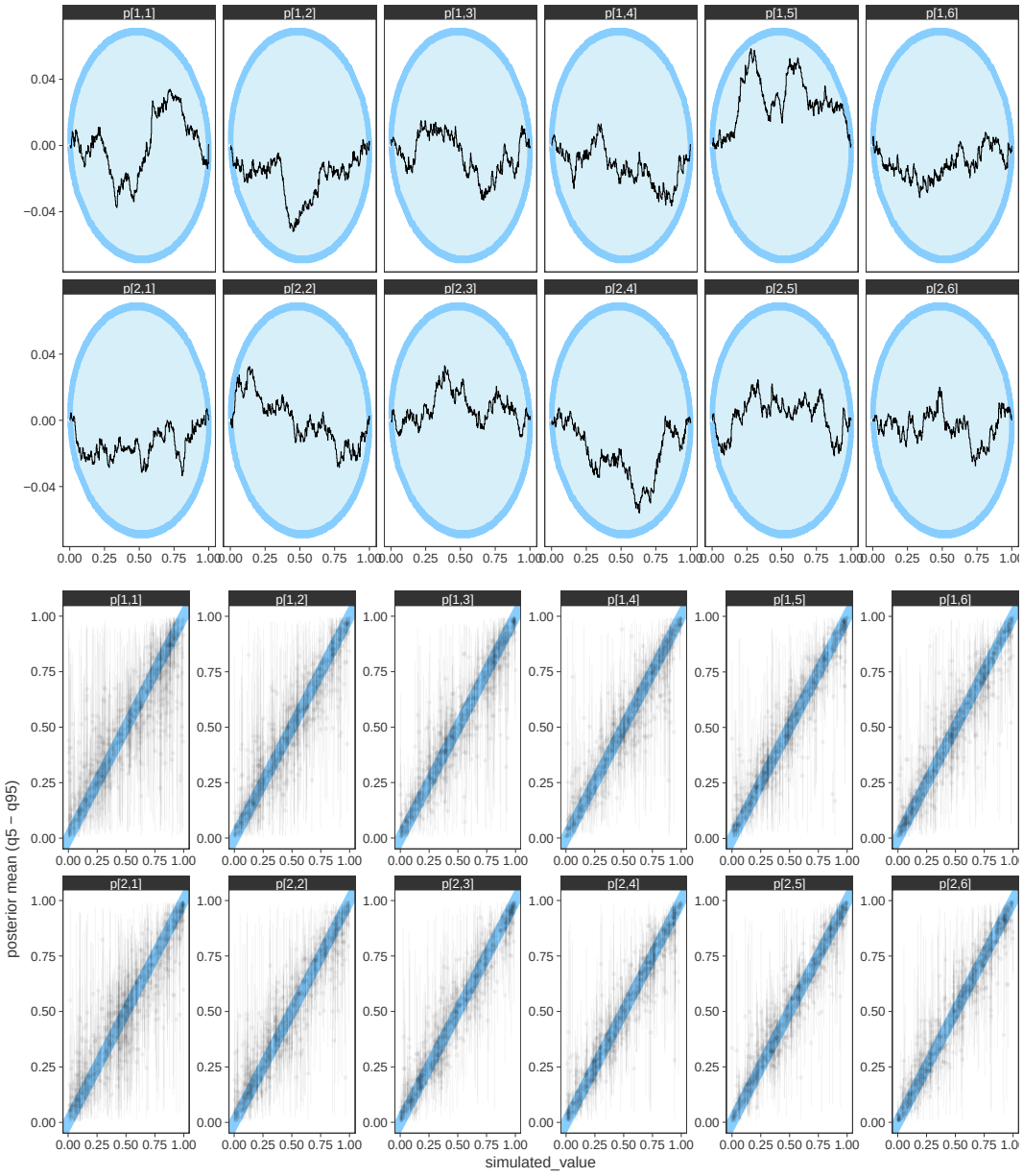
```
datasets <- SBC_generator_function(simulate_cmr, N_super = N_super, J = J,
                                   K_max = K_max, S = S, ME = T) |>
  generate_datasets(n_sims)
backend <- SBC_backend_cmdstan_sample(cmdstan_model(here("stan/cjs-me-rd.stan")),
                                      init = 0.1, chains = chains,
                                      iter_warmup = iter_warmup,
                                      iter_sampling = iter_sampling,
                                      max_treedepth = max_treedepth)
sbc <- compute_SBC(datasets, backend, cores_per_fit = chains, keep_fits = F)
```

Mean time for the slowest of 8 chains with 700 iterations: 0.147 minutes.

```
plot_sbc(sbc, c(str_c(rep(c("h", "eta"), each = S), "[", rep(1:S, 2), "]"),
                str_c("q[", 1:(S * (S - 1)), "]"),
                str_c("delta[", 1:(S - 1), "]")))
```



```
plot_sbc(sbc, str_c("p[", rep(1:S, each = J), ",", rep(1:J, S), "]"), ncol = J)
```



Jolly-Seber

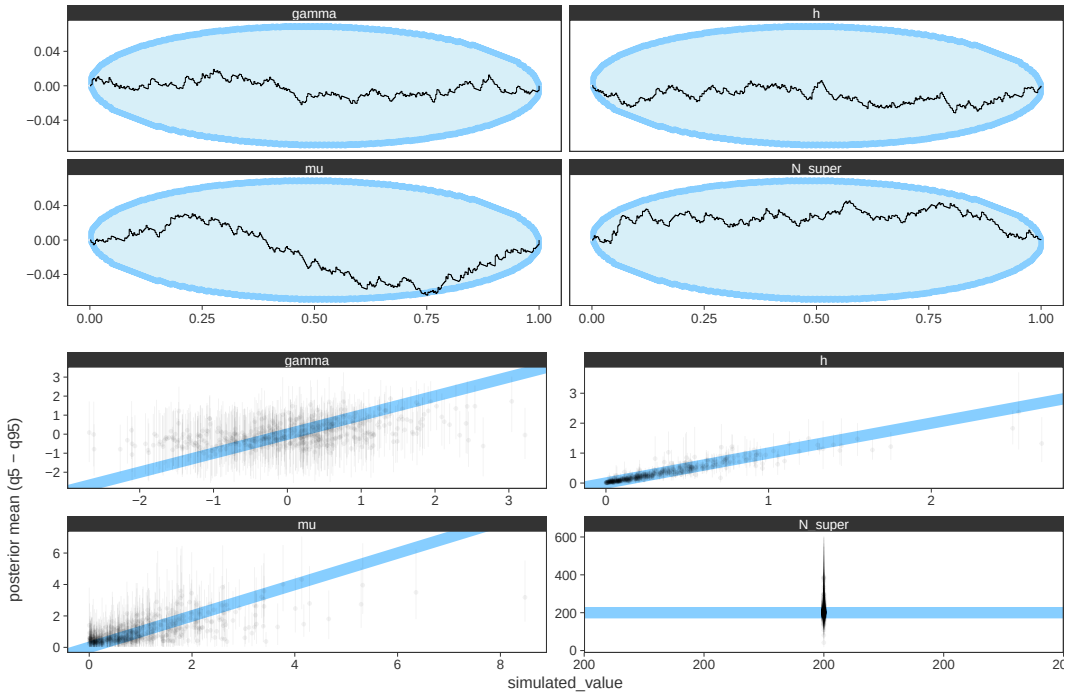
Single survey

```
datasets <- SBC_generator_function(simulate_cmr, N_super = N_super, J = J,
                                   JS = T) |>
  generate_datasets(n_sims)
backend <- SBC_backend_cmdstan_sample(cmdstan_model(here("stan/js.stan")),
                                      init = 0.1, chains = chains,
                                      iter_warmup = iter_warmup,
                                      iter_sampling = iter_sampling,
                                      max_treedepth = max_treedepth)
sbc <- compute_SBC(datasets, backend, cores_per_fit = chains, keep_fits = F)
```

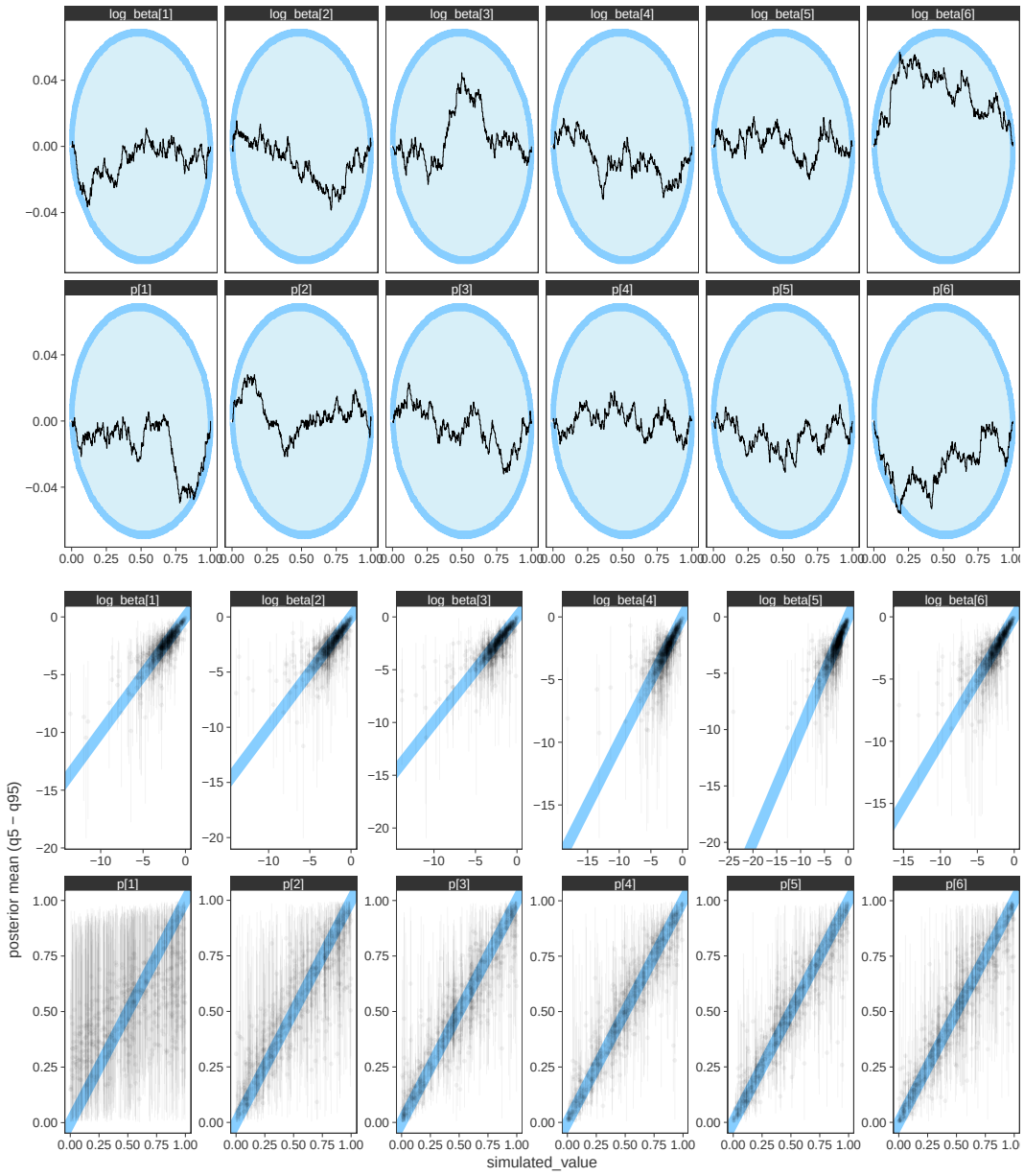
Mean time for the slowest of 8 chains with 700 iterations: 0.024 minutes.

> **Warning**
>
> In single survey Jolly-Seber models with time-varying entry and detection probabilities, the first entry and detection probabilities are confounded (Schwarz and Arnason 2008). Notice the difference in parameter estimates with the robust design.

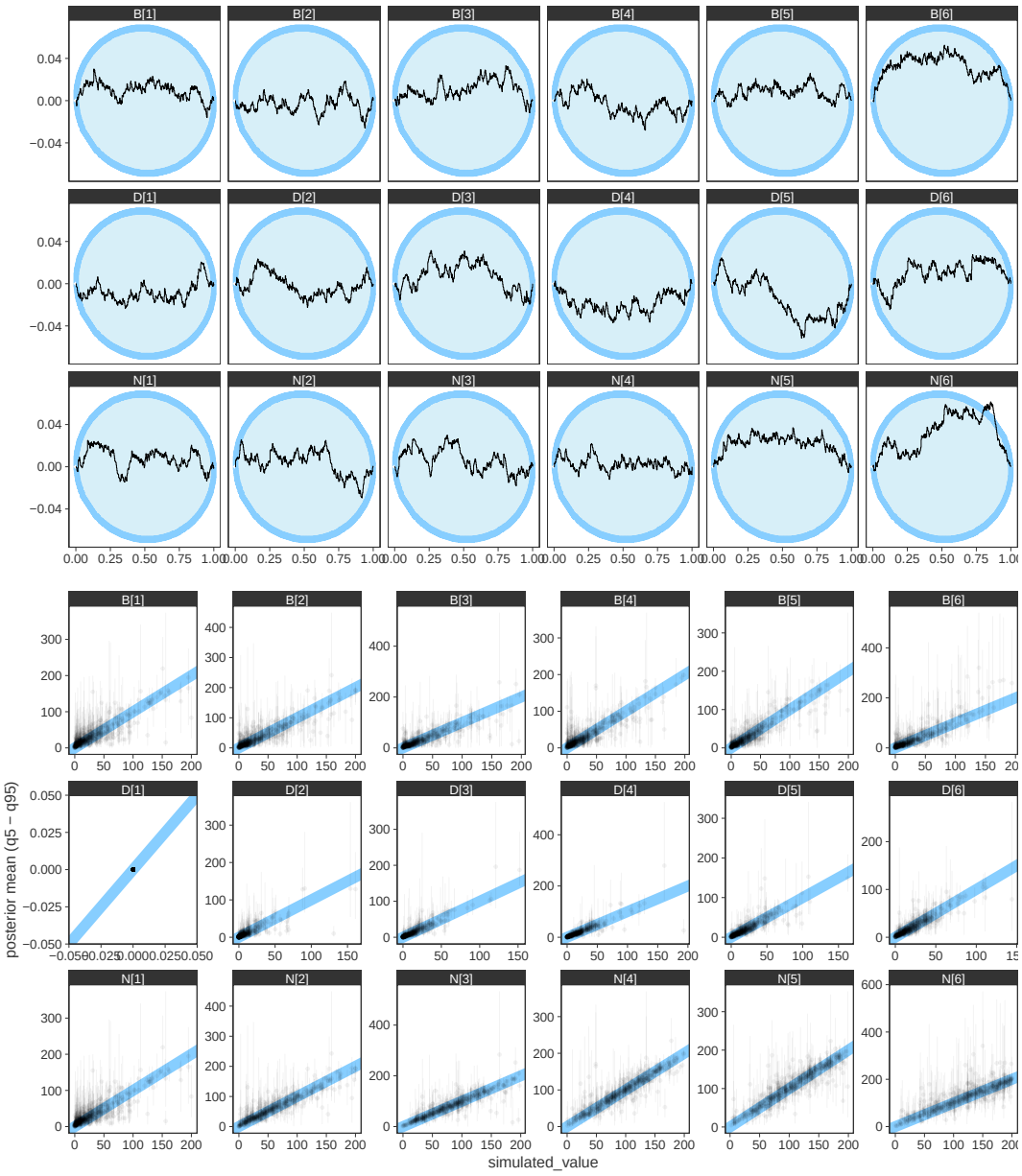
```
plot_sbc(sbc, c("h", "mu", "gamma", "N_super"))
```



```
plot_sbc(sbc, str_c(rep(c("log_beta", "p"), each = J), "[", rep(1:J, 2), "]"),
         ncol = J)
```



```
plot_sbc(sbc, str_c(rep(c("N", "B", "D"), each = J), "[", rep(1:J, 3), "]"),
         ncol = J)
```

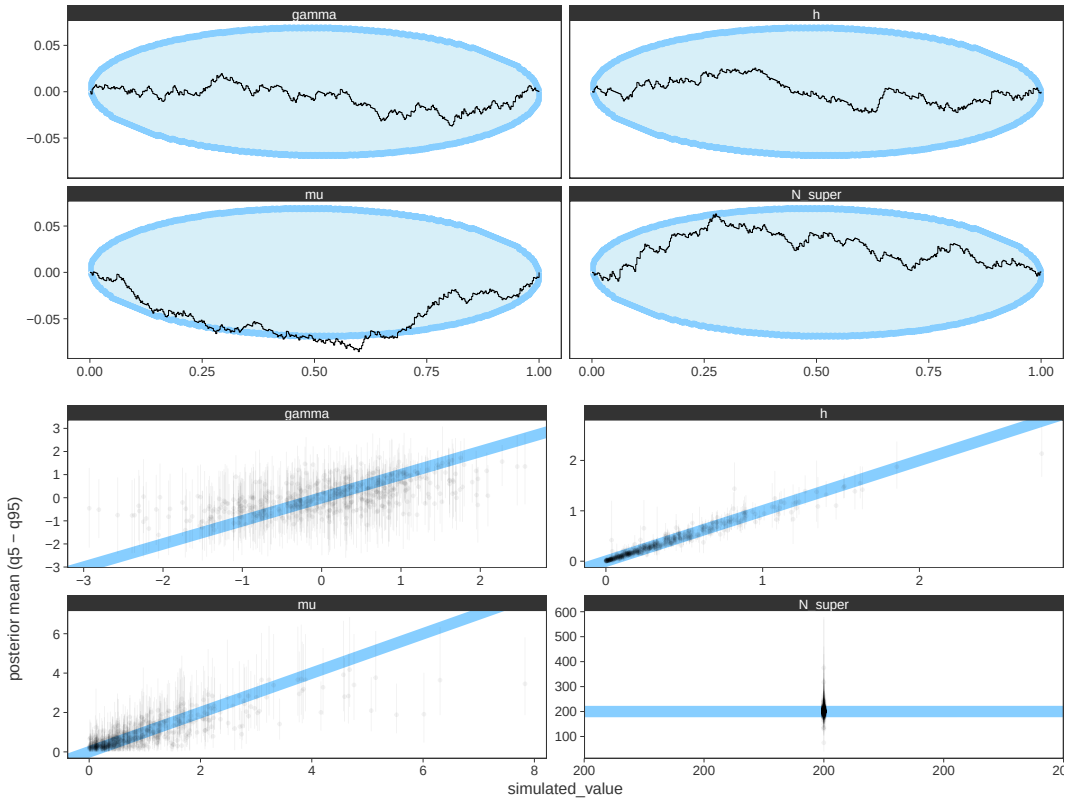


Robust design

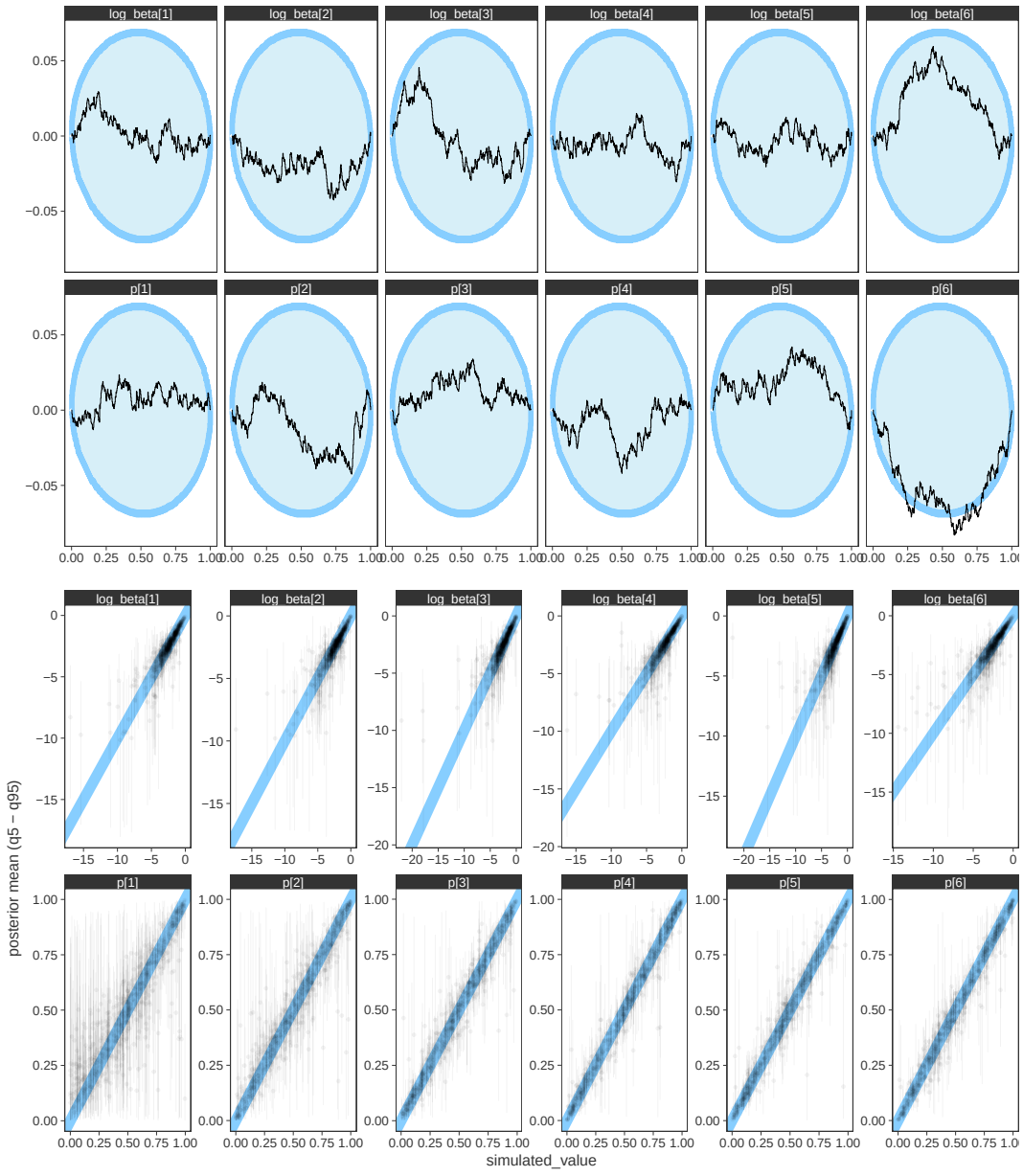
```
datasets <- SBC_generator_function(simulate_cmr, N_super = N_super, J = J,
                                   K_max = K_max, JS = T) |>
  generate_datasets(n_sims)
backend <- SBC_backend_cmdstan_sample(cmdstan_model(here("stan/js-rd.stan")),
                                      init = 0.1, chains = chains,
                                      iter_warmup = iter_warmup,
                                      iter_sampling = iter_sampling,
                                      max_treedepth = max_treedepth)
sbc <- compute_SBC(datasets, backend, cores_per_fit = chains, keep_fits = F)
```

Mean time for the slowest of 8 chains with 700 iterations: 0.04 minutes.

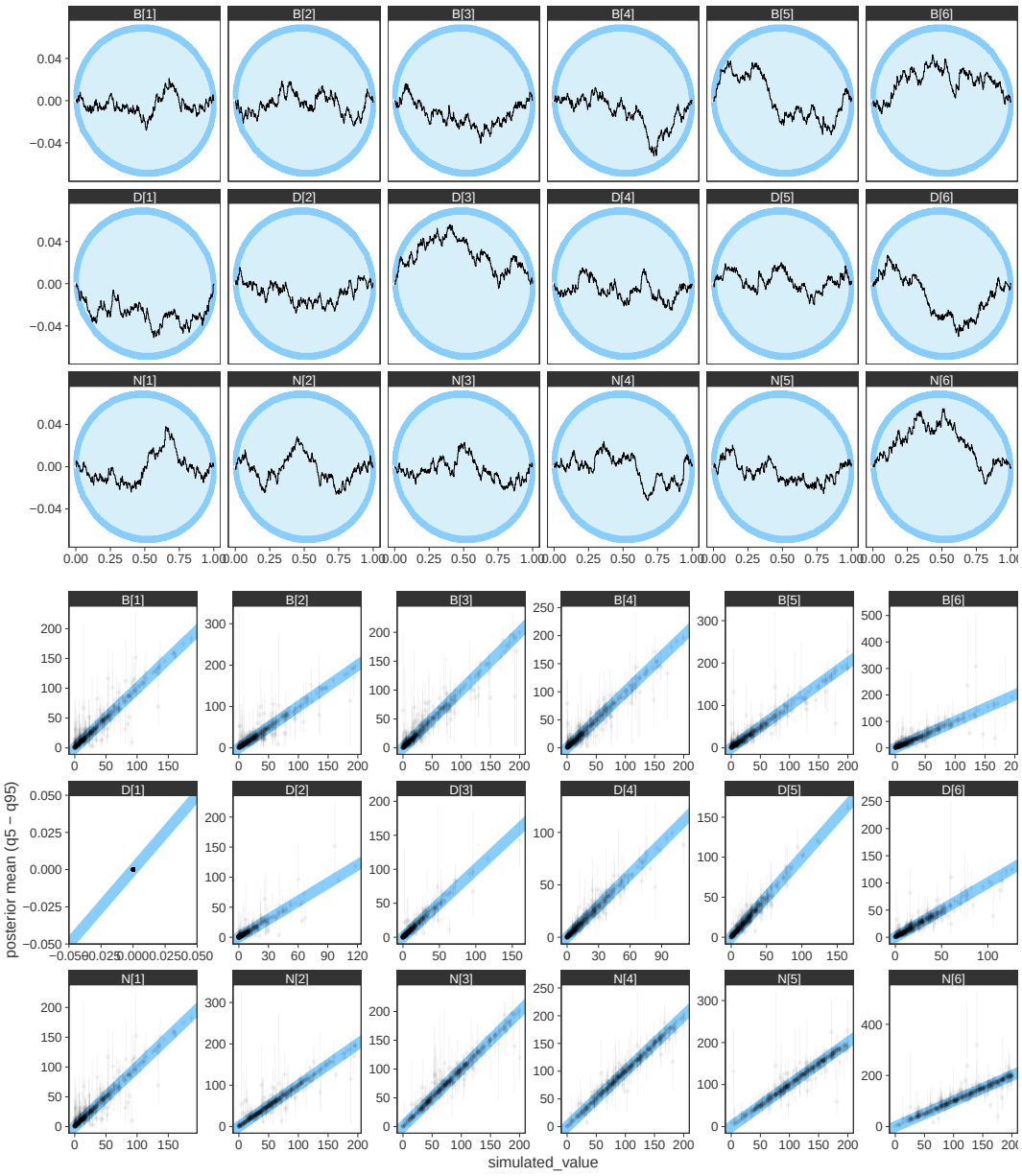
```
plot_sbc(sbc, c("h", "mu", "gamma", "N_super"))
```



```
plot_sbc(sbc, str_c(rep(c("log_beta", "p"), each = J), "[", rep(1:J, 2), "]"),
         ncol = J)
```



```
plot_sbc(sbc, str_c(rep(c("N", "B", "D"), each = J), "[", rep(1:J, 3), "]"),
         ncol = J)
```



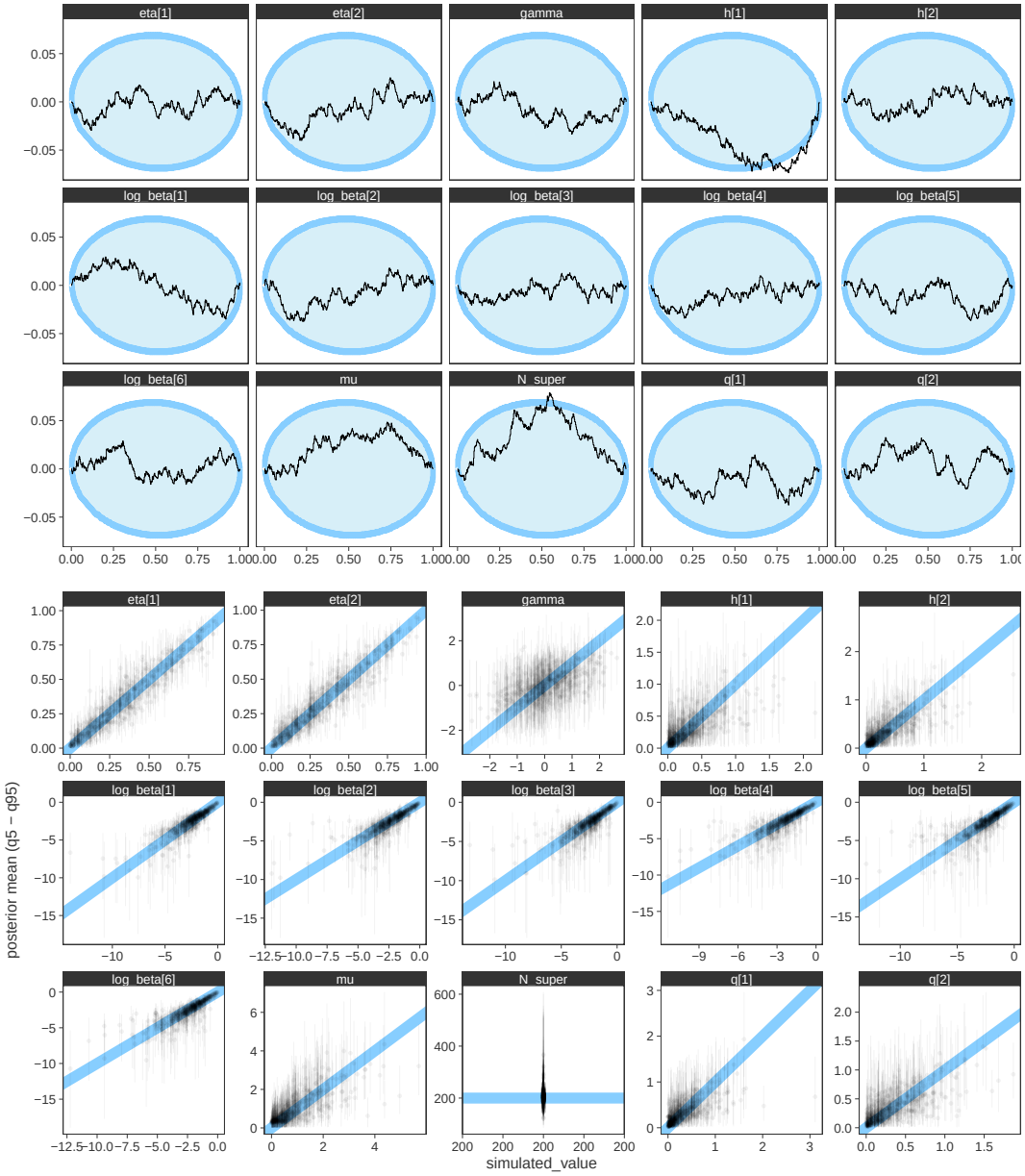
Multistate Jolly-Seber

Single survey

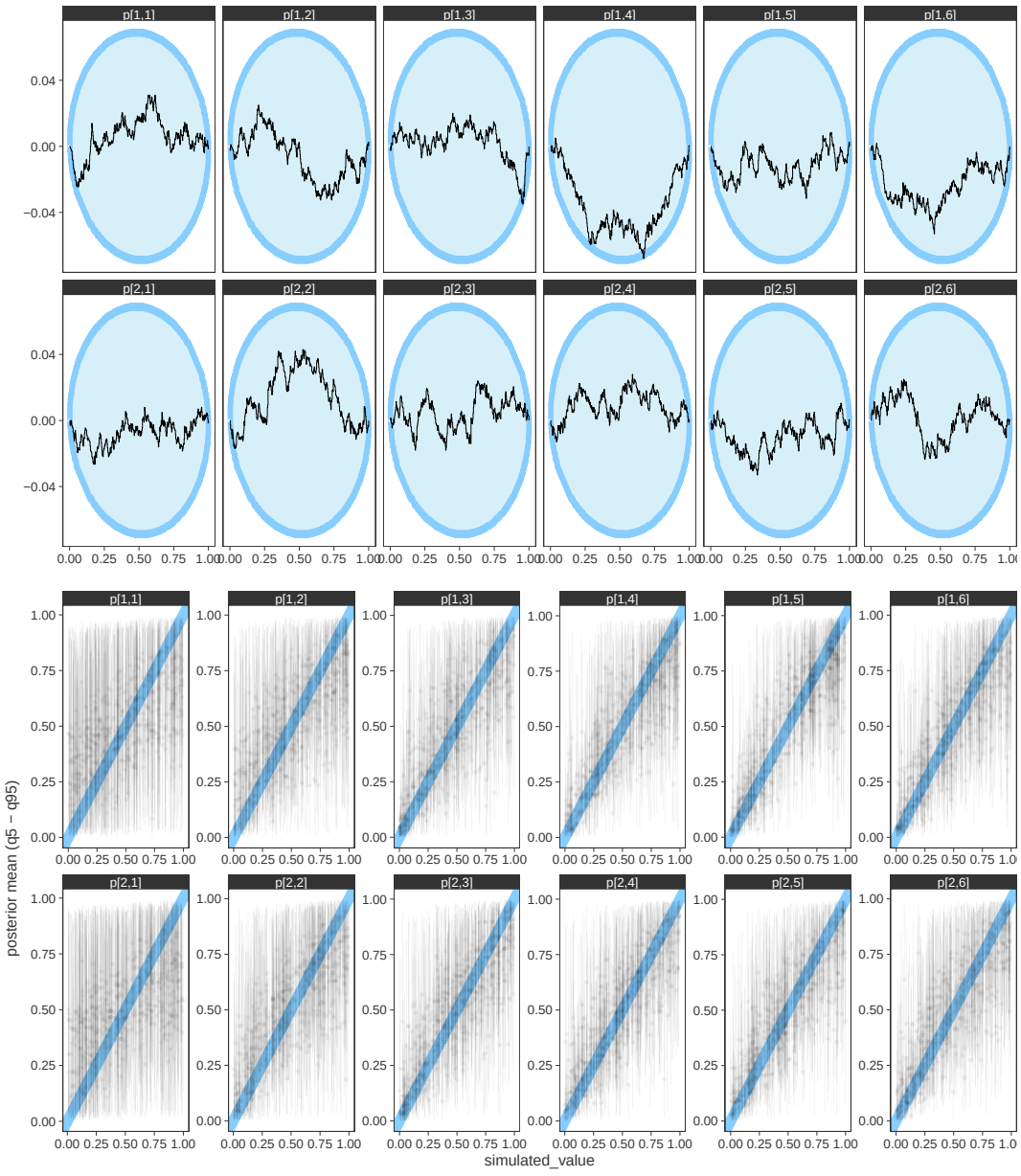
```
datasets <- SBC_generator_function(simulate_cmr, N_super = N_super, J = J,
                                   S = 3, JS = T) |>
  generate_datasets(n_sims)
backend <- SBC_backend_cmdstan_sample(cmdstan_model(here("stan/js-ms.stan")),
                                      init = 0.1, chains = chains,
                                      iter_warmup = iter_warmup,
                                      iter_sampling = iter_sampling,
                                      max_treedepth = max_treedepth)
sbc <- compute_SBC(datasets, backend, cores_per_fit = chains, keep_fits = F)
```

Mean time for the slowest of 8 chains with 700 iterations: 0.198 minutes.

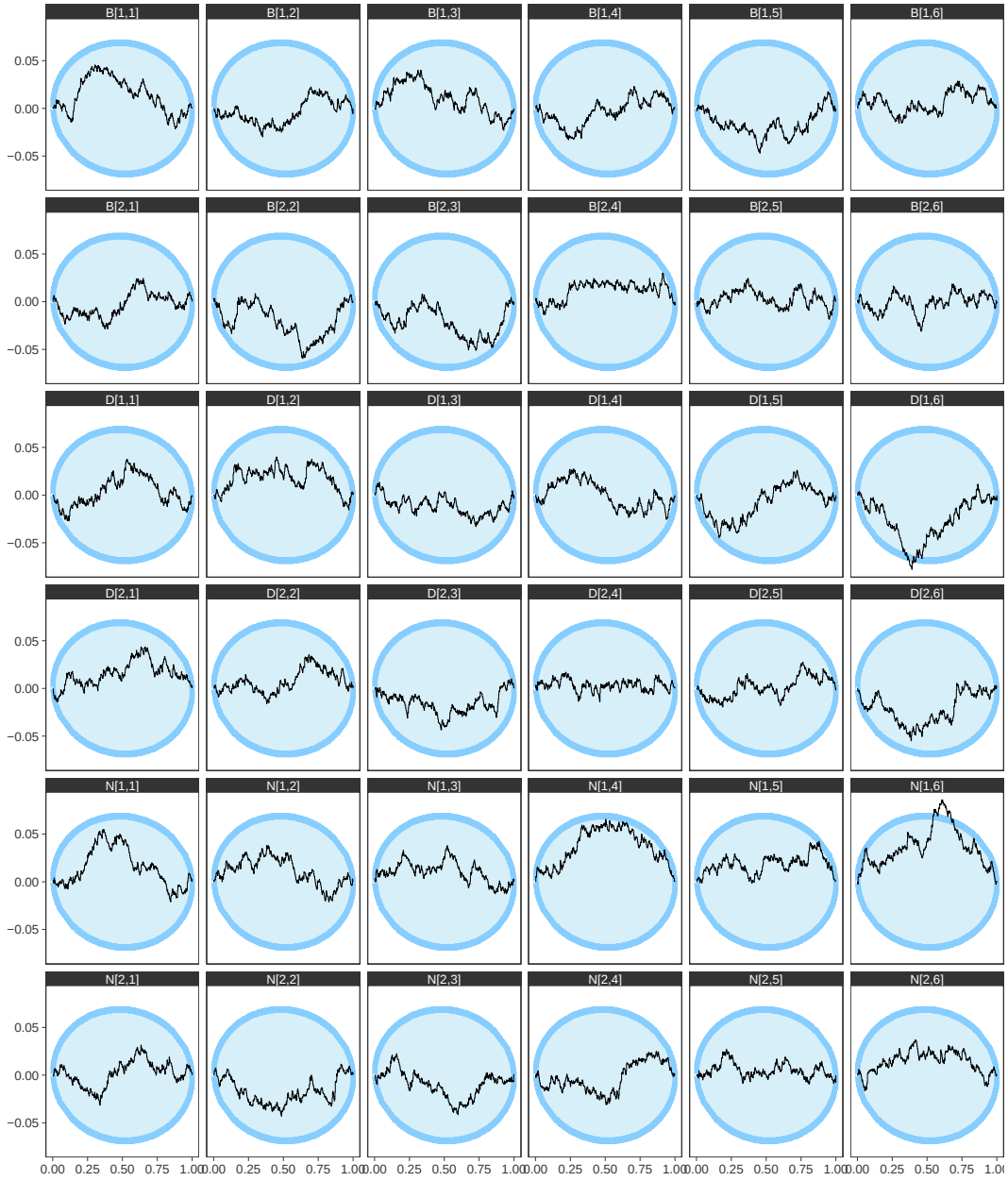
```
plot_sbc(sbc, c(str_c(rep(c("h", "eta"), each = S), "[", rep(1:S, 2), "]"),
                str_c("q[", 1:(S * (S - 1)), "]"),
                str_c("log_beta[", 1:J, "]"), "mu", "gamma", "N_super"),
         nrow = 3)
```



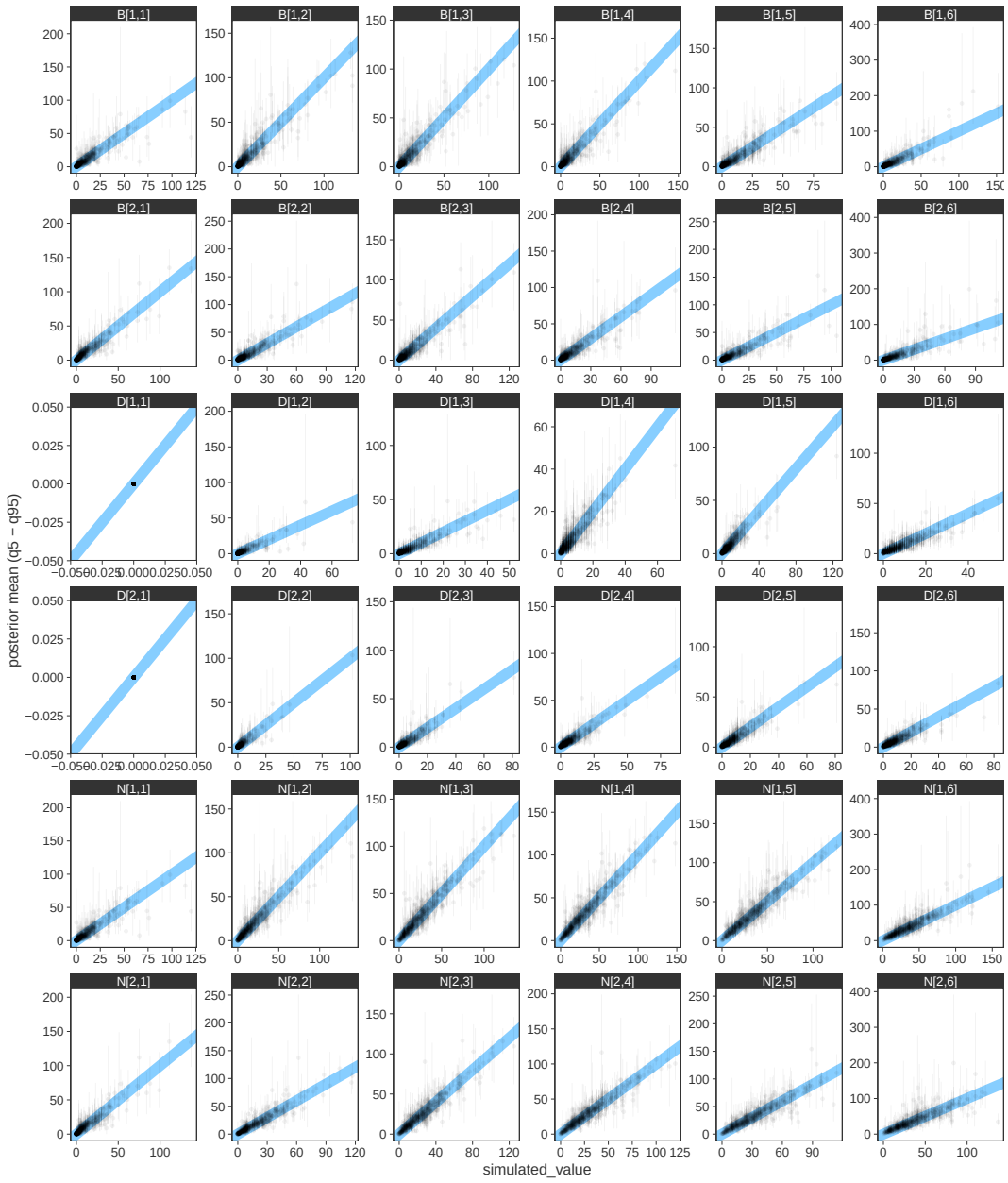
```
params <- map(c("p", "N", "B", "D"),
              ~str_c(., "[", rep(1:S, each = J), ",", rep(1:J, S), "]"))
plot_sbc(sbc, params[[1]], nrow = 2)
```



```
plot_ecdf_diff(sbc, flatten_chr(params[2:4])) +
  facet_wrap(~ variable, ncol = J) +
  theme(legend.position = "none")
```



```
plot_sim_estimated(sbc, flatten_chr(params[2:4])) +
  facet_wrap(~ variable, ncol = J, scales = "free")
```

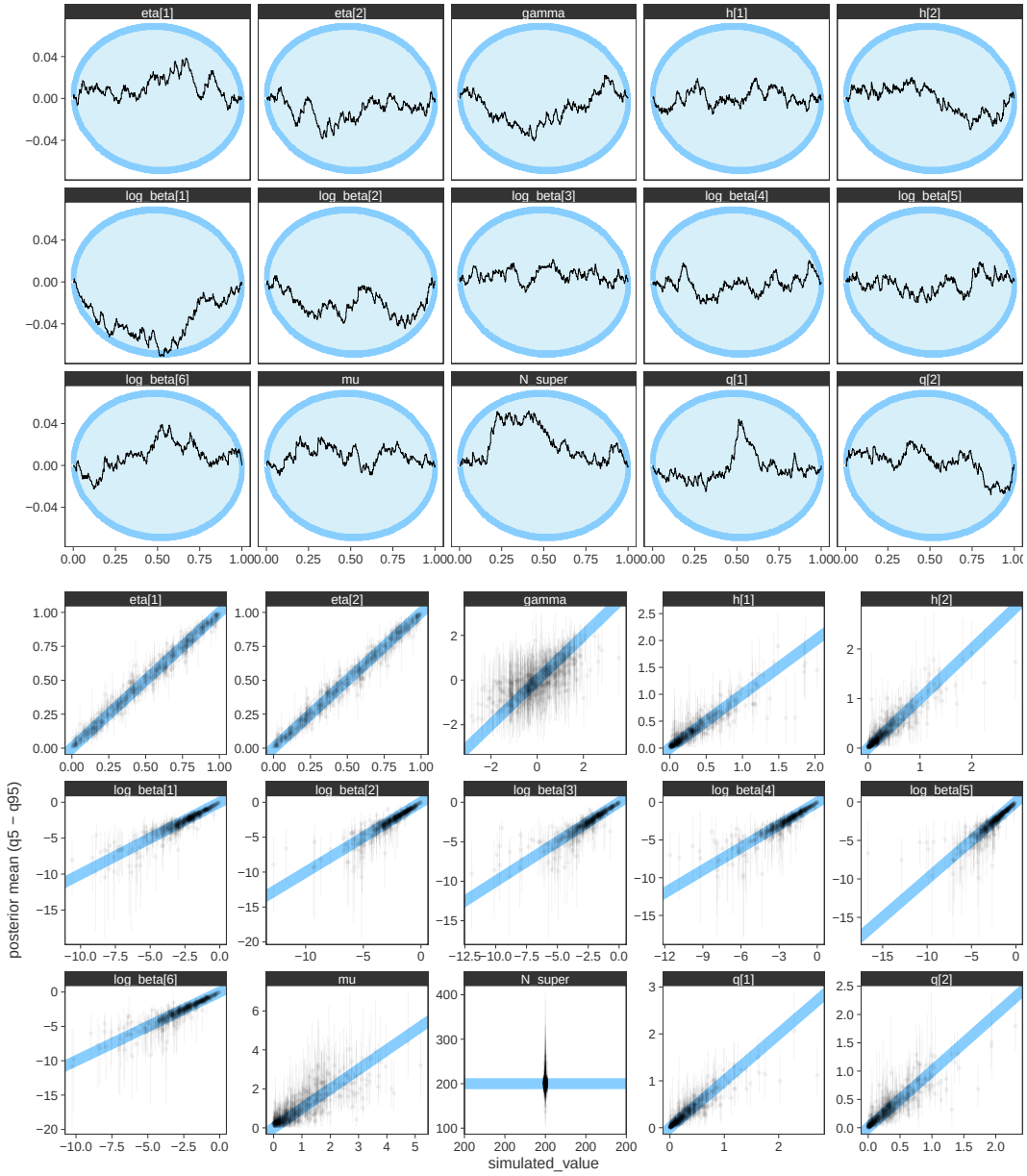


Robust design

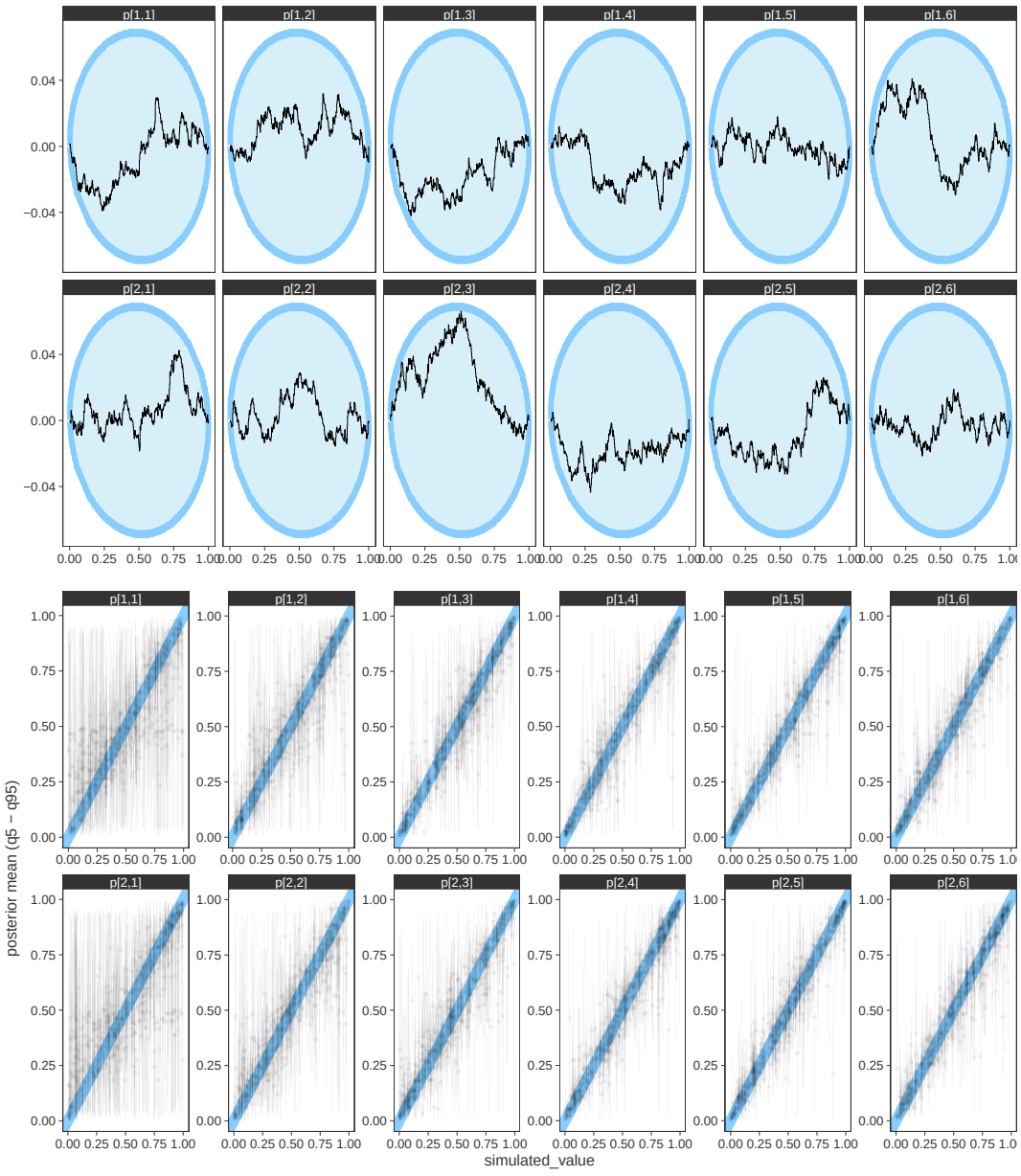
```
datasets <- SBC_generator_function(simulate_cmr, N_super = N_super, J = J,
                                   K_max = K_max, S = S, JS = T) |>
  generate_datasets(n_sims)
backend <- SBC_backend_cmdstan_sample(cmdstan_model(here("stan/js-ms-rd.stan")),
                                      init = 0.1, chains = chains,
                                      iter_warmup = iter_warmup,
                                      iter_sampling = iter_sampling,
                                      max_treedepth = max_treedepth)
sbc <- compute_SBC(datasets, backend, cores_per_fit = chains, keep_fits = F)
```

Mean time for the slowest of 8 chains with 700 iterations: 0.142 minutes.

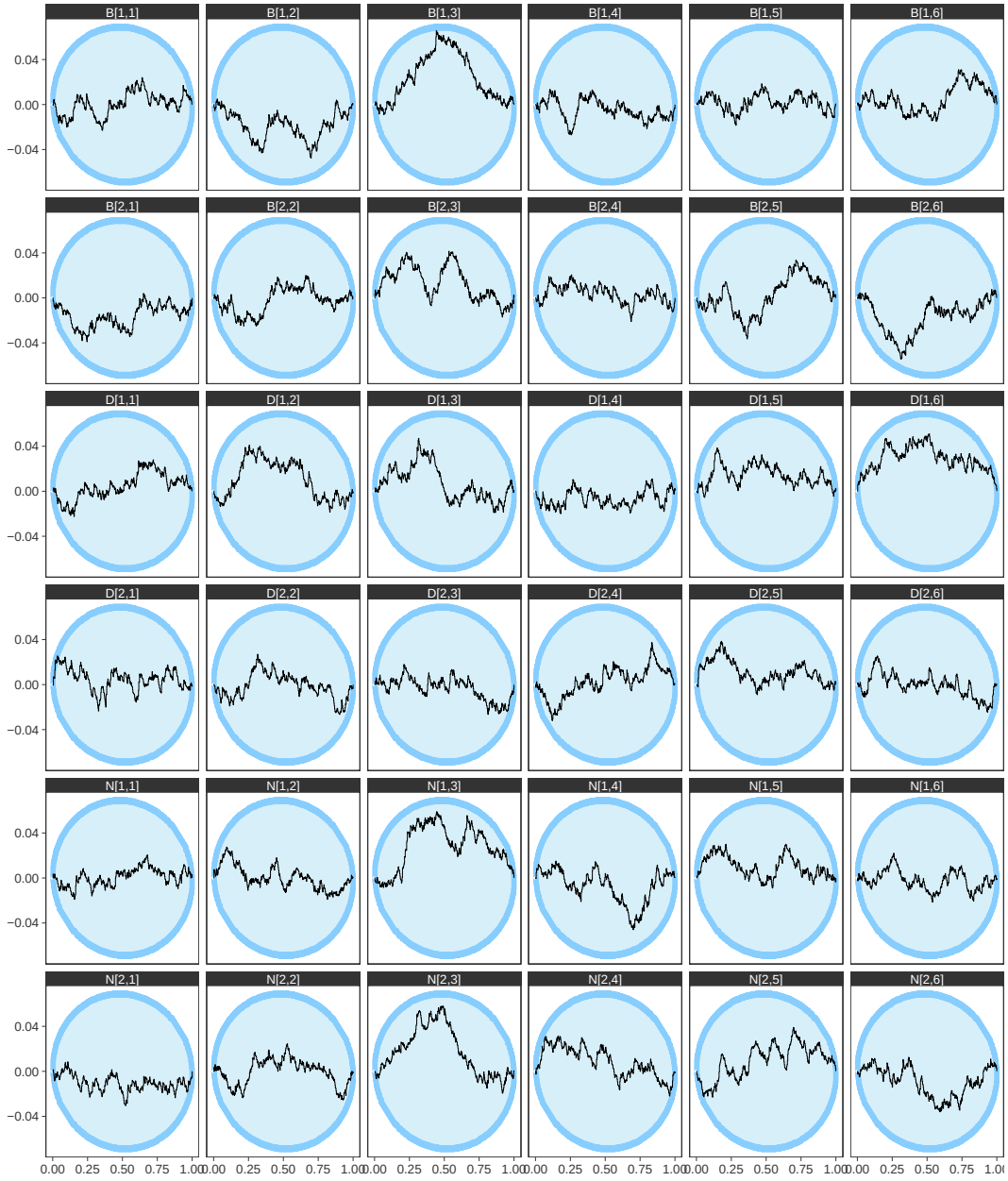
```
plot_sbc(sbc, c(str_c(rep(c("h", "eta"), each = S), "[", rep(1:S, 2), "]"),
                str_c("q[", 1:(S * (S - 1)), "]"),
                str_c("log_beta[", 1:J, "]"), "mu", "gamma", "N_super"),
         nrow = 3)
```



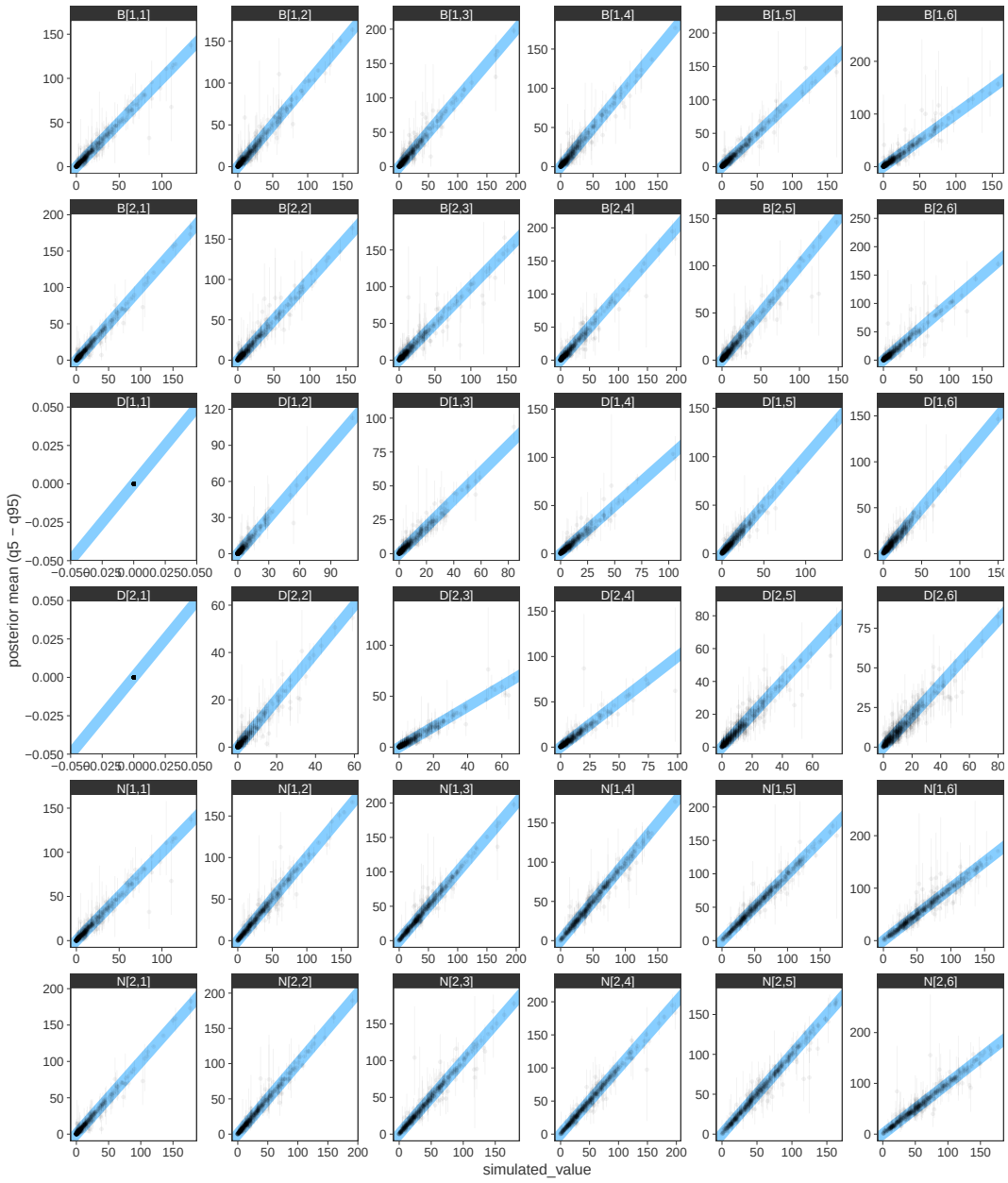
```
params <- map(c("p", "N", "B", "D"),
              ~str_c(., "[", rep(1:S, each = J), ",", rep(1:J, S), "]"))
plot_sbc(sbc, params[[1]], nrow = 2)
```



```
plot_ecdf_diff(sbc, flatten_chr(params[2:4])) +
  facet_wrap(~ variable, ncol = J) +
  theme(legend.position = "none")
```



```
plot_sim_estimated(sbc, flatten_chr(params[2:4])) +
  facet_wrap(~ variable, ncol = J, scales = "free")
```



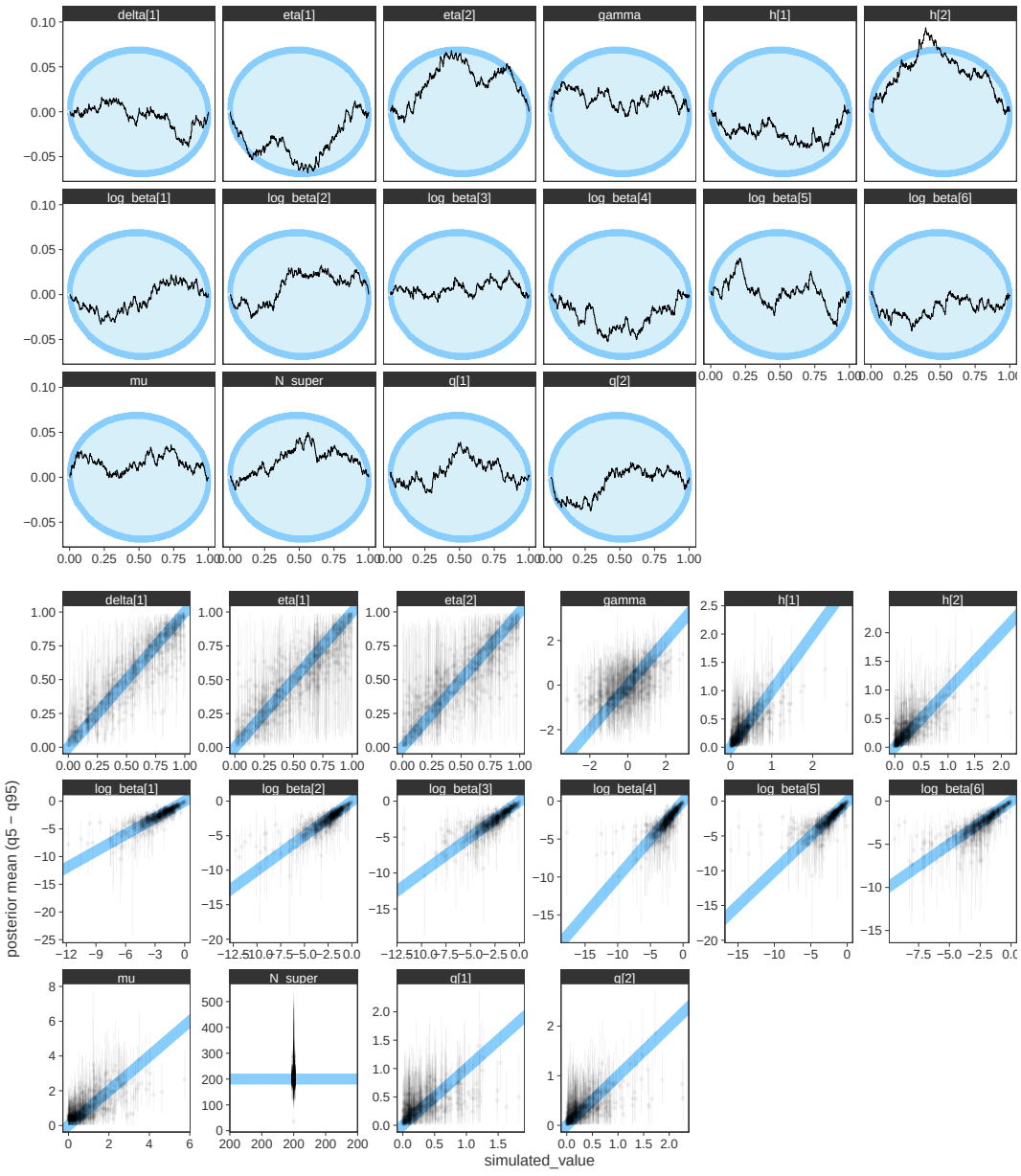
Multievent Jolly-Seber

Single survey

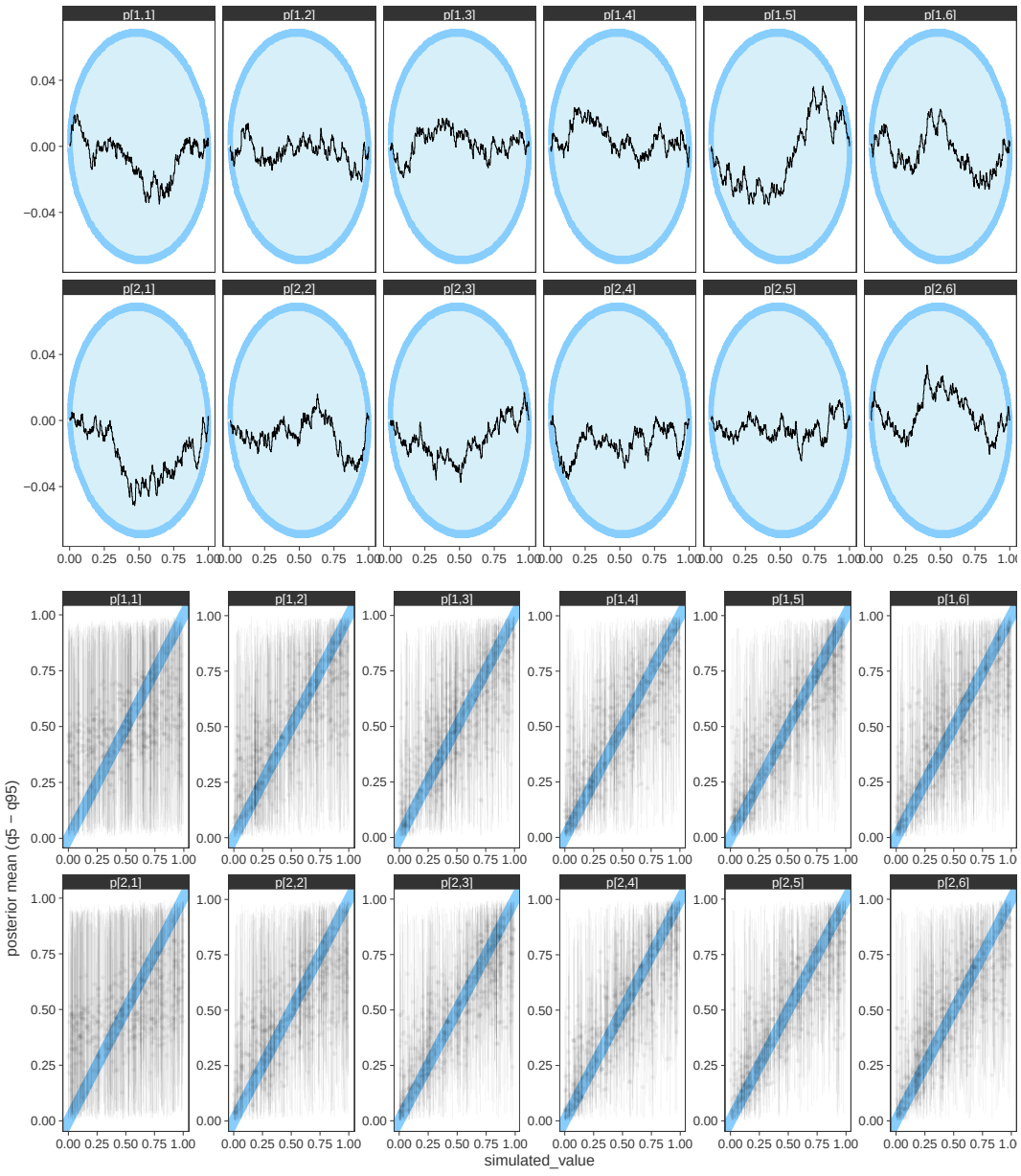
```
datasets <- SBC_generator_function(simulate_cmr, N_super = N_super, J = J,
                                   S = S, ME = T, JS = T) |>
  generate_datasets(n_sims)
backend <- SBC_backend_cmdstan_sample(cmdstan_model(here("stan/js-me.stan")),
                                      init = 0.1, chains = chains,
                                      iter_warmup = iter_warmup,
                                      iter_sampling = iter_sampling,
                                      max_treedepth = max_treedepth)
sbc <- compute_SBC(datasets, backend, cores_per_fit = chains, keep_fits = F)
```

Mean time for the slowest of 8 chains with 700 iterations: 0.808 minutes.

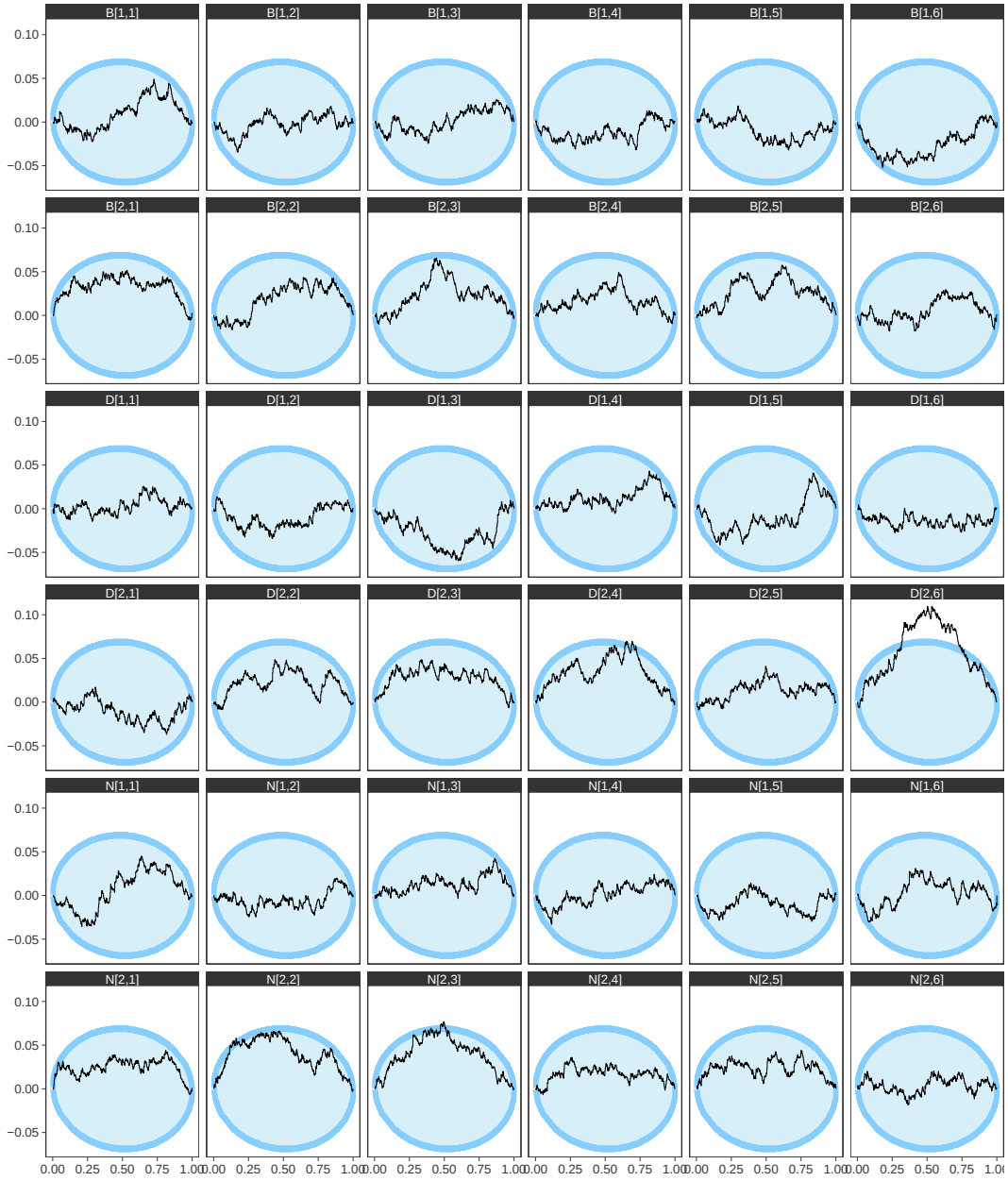
```
plot_sbc(sbc, c(str_c(rep(c("h", "eta"), each = S), "[", rep(1:S, 2), "]"),
                str_c("q[", 1:(S * (S - 1)), "]"),
                str_c("log_beta[", 1:J, "]"), str_c("delta[", 1:(S - 1), "]"),
                "mu", "gamma", "N_super"), nrow = 3)
```



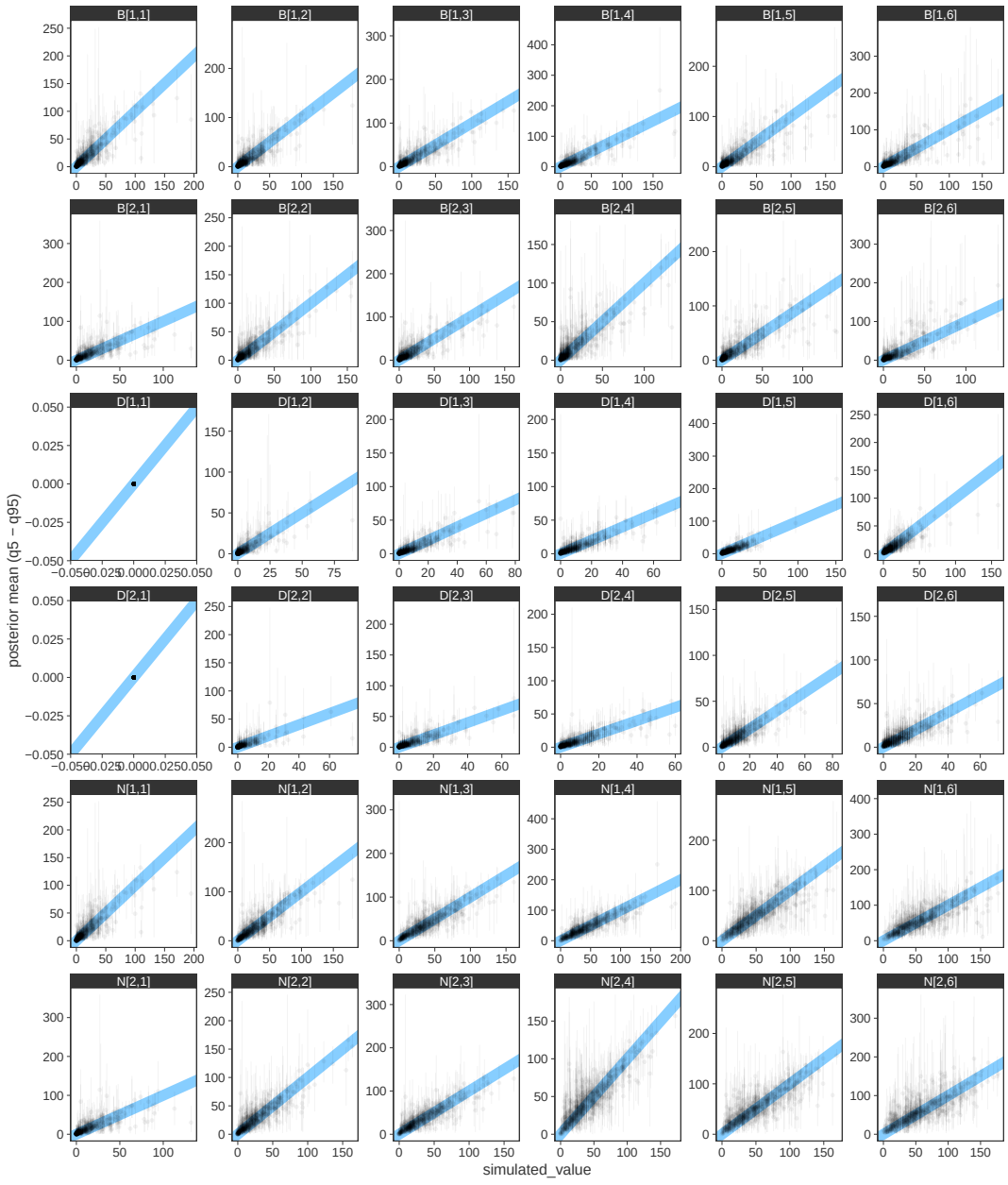
```
params <- map(c("p", "N", "B", "D"),
              ~str_c(., "[", rep(1:S, each = J), ",", rep(1:J, S), "]"))
plot_sbc(sbc, params[[1]], nrow = 2)
```



```
plot_ecdf_diff(sbc, flatten_chr(params[2:4])) +
  facet_wrap(~ variable, ncol = J) +
  theme(legend.position = "none")
```



```
plot_sim_estimated(sbc, flatten_chr(params[2:4])) +
  facet_wrap(~ variable, ncol = J, scales = "free")
```

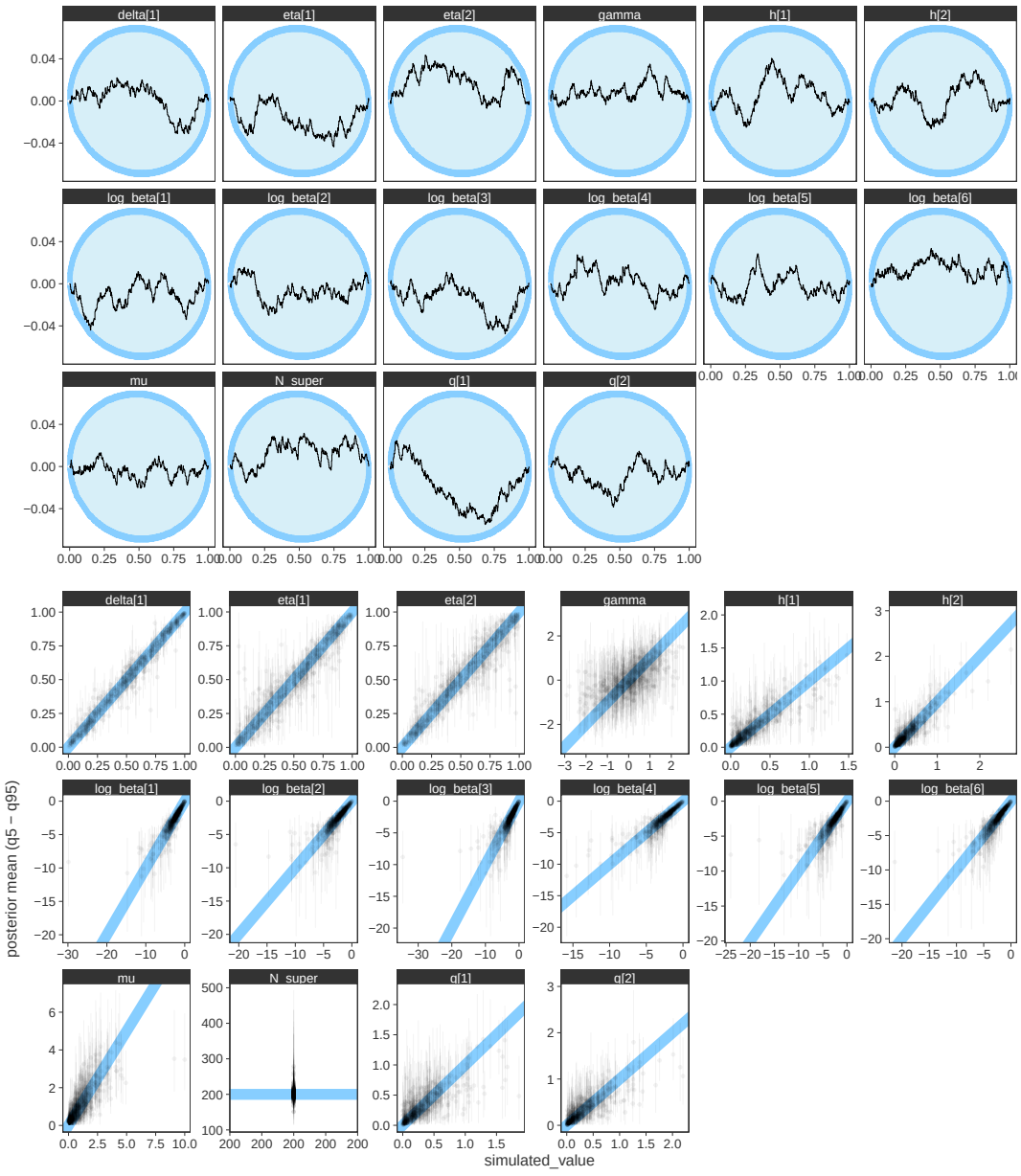


Robust design

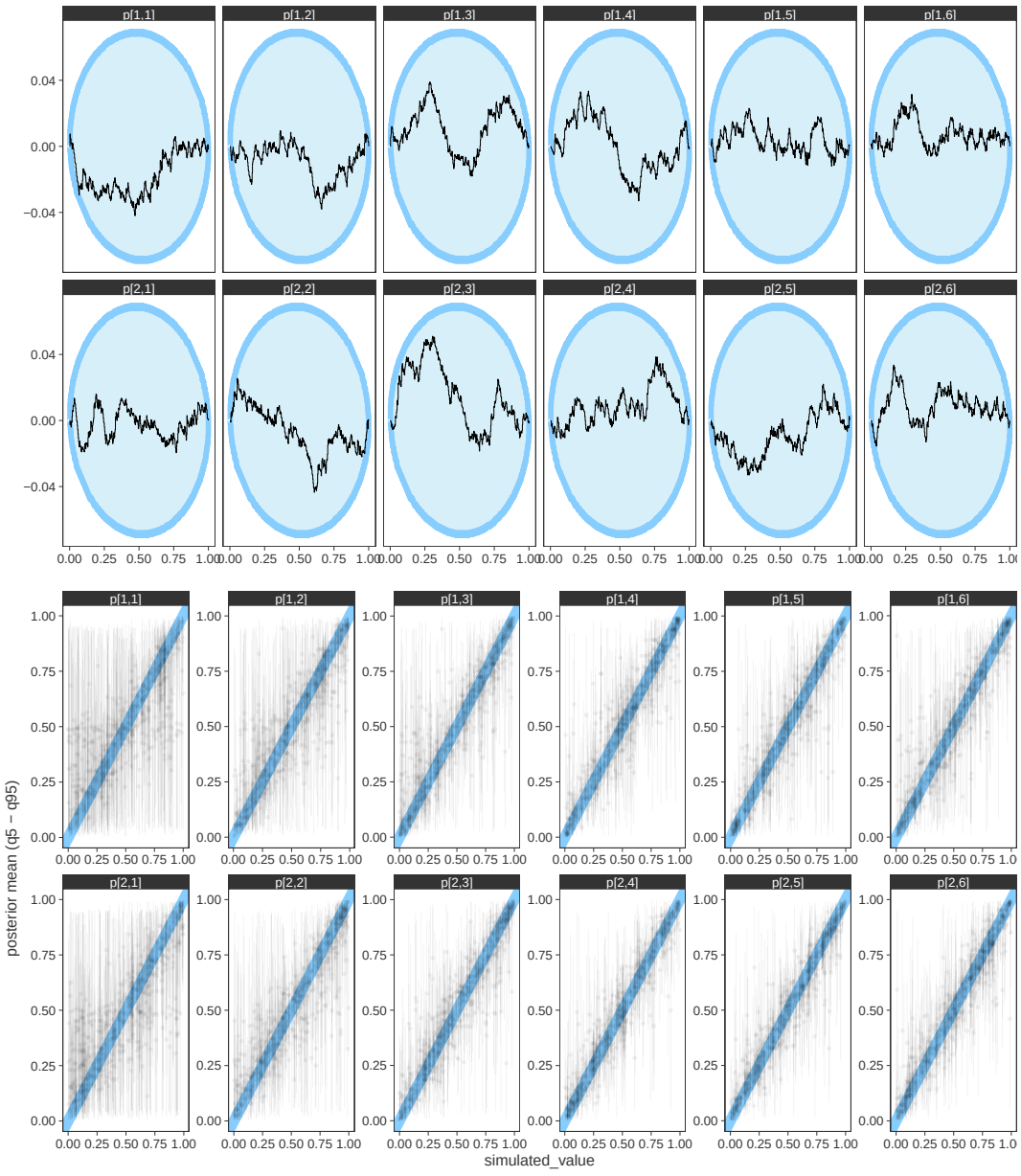
```
datasets <- SBC_generator_function(simulate_cmr, N_super = N_super, J = J,
                                   K_max = K_max, S = S, ME = T, JS = T) |>
  generate_datasets(n_sims)
backend <- SBC_backend_cmdstan_sample(cmdstan_model(here("stan/js-me-rd.stan")),
                                      init = 0.1, chains = chains,
                                      iter_warmup = iter_warmup,
                                      iter_sampling = iter_sampling,
                                      max_treedepth = max_treedepth)
sbc <- compute_SBC(datasets, backend, cores_per_fit = chains, keep_fits = F)
```

Mean time for the slowest of 8 chains with 700 iterations: 0.813 minutes.

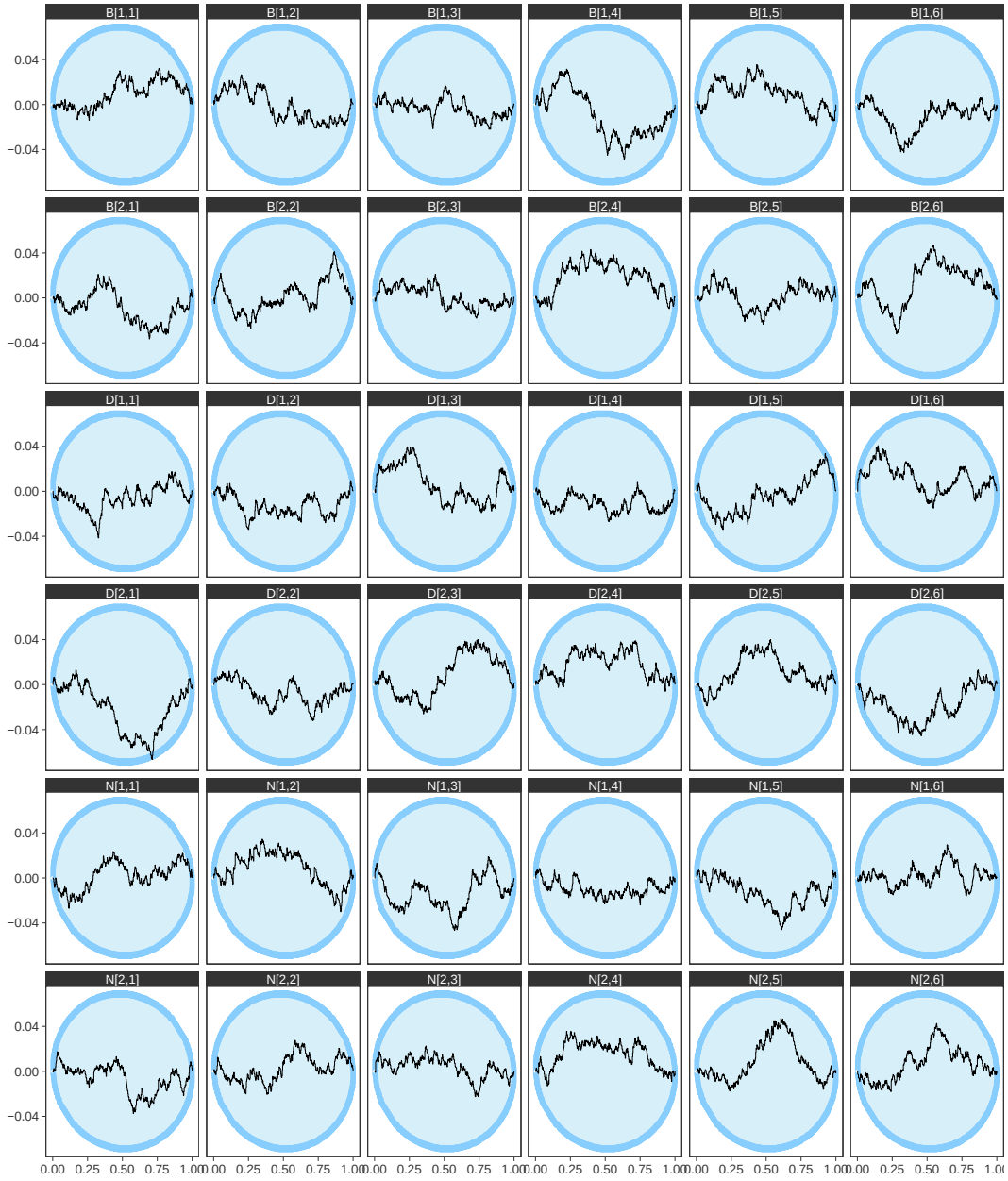
```
plot_sbc(sbc, c(str_c(rep(c("h", "eta"), each = S), "[", rep(1:S, 2), "]"),
                str_c("q[", 1:(S * (S - 1)), "]"),
                str_c("log_beta[", 1:J, "]"), str_c("delta[", 1:(S - 1), "]"),
                "mu", "gamma", "N_super"), nrow = 3)
```



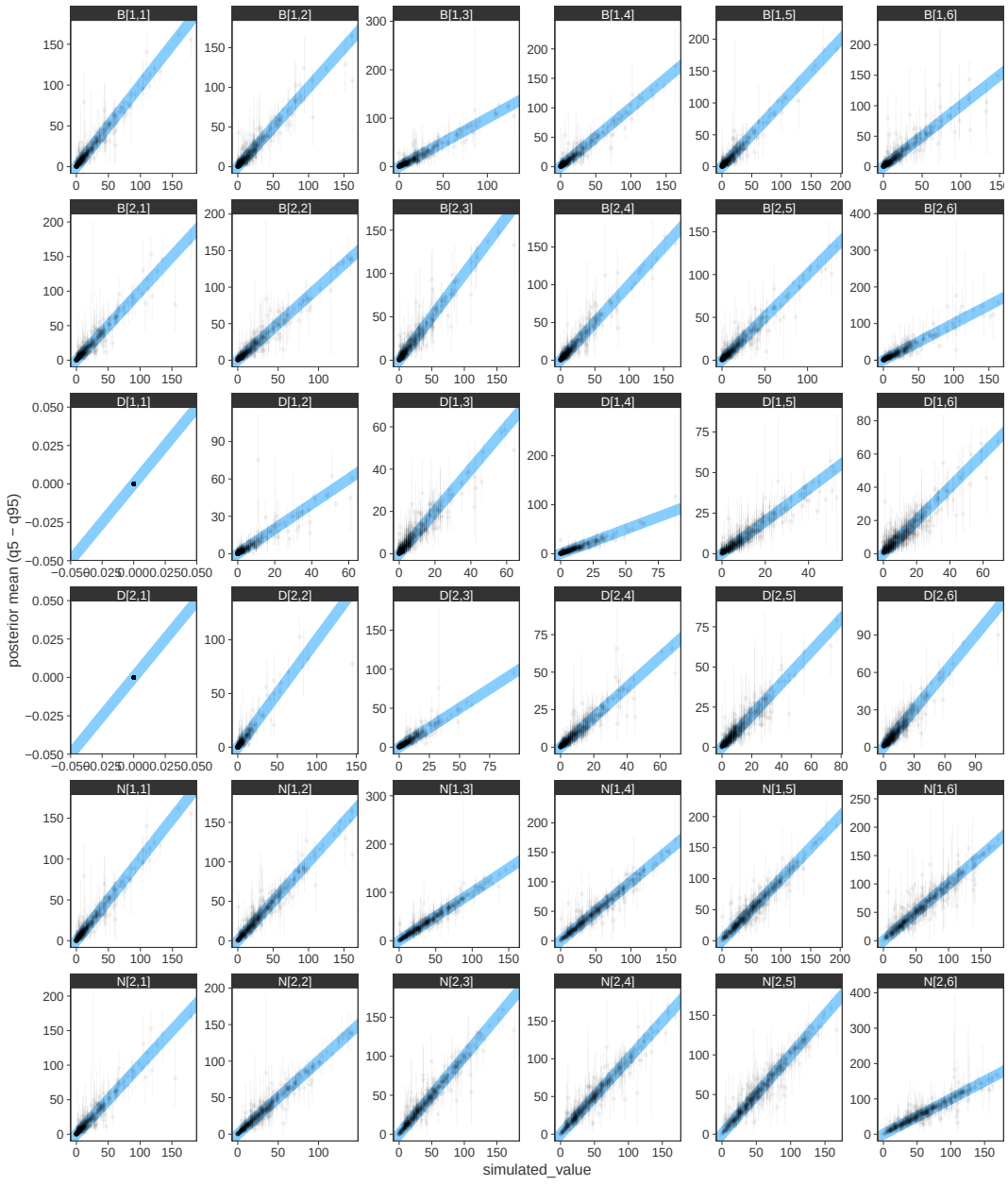
```
params <- map(c("p", "N", "B", "D"),
              ~str_c(., "[", rep(1:S, each = J), ",", rep(1:J, S), "]"))
plot_sbc(sbc, params[[1]], nrow = 2)
```



```
plot_ecdf_diff(sbc, flatten_chr(params[2:4])) +
  facet_wrap(~ variable, ncol = J) +
  theme(legend.position = "none")
```



```
plot_sim_estimated(sbc, flatten_chr(params[2:4])) +
  facet_wrap(~ variable, ncol = J, scales = "free")
```



Simulation script

```
 if (!require(expm)) install.packages("expm")
simulate_cmr <- function(N_super = 100, J = 8, K_max = 1, S = 1, I_aug = 500,
                         ME = FALSE, JS = FALSE, dirichlet = FALSE,
                         ind = FALSE, collapse = FALSE, grainsize = 0,
                         mu_gamma_prior = c(1, 1), gamma_normal_prior = c(0, 1),
                         h_gamma_prior = c(1, 3), q_gamma_prior = c(1, 3),
                         p_beta_prior = c(1, 1), eta_dirichlet_prior = 1,
                         delta_beta_prior = c(1, 1)) {

  # transformed data and parameters
  Jm1 <- J - 1
  if (K_max > 1) {
    K <- sample(1:K_max, J, replace = T, prob = sort(runif(K_max)))
    while (K[1] == 1) {
      K[1] <- sample(2:K_max, 1)
    }
    K_max <- max(K)
  } else {
    K <- rep(1, J)
  }
  tau <- rlnorm(Jm1)
  tau_scl <- tau / exp(mean(log(tau)))
  h <- rgamma(S, h_gamma_prior[1], h_gamma_prior[2])
  p <- matrix(rbeta(S * J, p_beta_prior[1], p_beta_prior[2]), S, J)
  if (S == 1) {
    phi_tau <- exp(-h * tau)
  } else {
    Sm1 <- S - 1 ; Sp1 <- S + 1
    q <- rgamma(S * Sm1, q_gamma_prior[1], q_gamma_prior[2])
    H <- array(0, c(Jm1, Sp1, Sp1))
    for (j in 1:Jm1) {
      H[j, , ] <- expm::expm(rate_matrix(h, q) * tau_scl[j])
    }
    if (ME) {
      delta <- rbeta(Sm1, delta_beta_prior[1], delta_beta_prior[2])
      E <- triangular_bidiagonal_stochastic_matrix(delta)
    }
    if (ME || JS) {
      if (length(eta_dirichlet_prior) == 1) {
```

```
      alpha <- rep(eta_dirichlet_prior, S)
    } else {
      alpha <- eta_dirichlet_prior
    }
    eta <- rdirch(1, alpha)
  }
}

# latent states and detection history
z <- matrix(0, N_super, J)
y <- array(0, c(N_super, J, K_max))

# Jolly-Seber
if (JS) {
  mu <- rgamma(1, mu_gamma_prior[1], mu_gamma_prior[2])
  gamma <- rnorm(1, gamma_normal_prior[1], gamma_normal_prior[2])
  log_alpha <- c(gamma, log(tau_scl))
  if (dirichlet) {
    beta <- rdirch(1, exp(log(mu) + log_alpha))
  } else {
    u <- rnorm(J, log_alpha, mu)
    beta <- softmax(u - mean(u))
  }
  b <- sort(rcat(N_super, beta))

  # single state ecological process
  if (S == 1) {
    B <- D <- numeric(J)
    for (i in 1:N_super) {
      b_i <- b[i]
      z[i, b_i] <- 1
      B[b_i] <- B[b_i] + 1
      if (b_i < J) {
        for (j in (b_i + 1):J) {
          jm1 <- j - 1
          z[i, j] <- rbinom(1, 1, z[i, jm1] * phi_tau[jm1])
          if (z[i, jm1] == 1 & z[i, j] == 0) {
            D[j] <- D[j] + 1
          }
        }
      }
```

```
      # observation process
      for (j in b_i:J) {
        y[i, j, 1:K[j]] <- rbinom(K[j], 1, z[i, j] * p[j])
      }
    }

    # multistate ecological process
  } else {
    B <- D <- matrix(0, S, J)
    for (i in 1:N_super) {
      b_i <- b[i]
      z[i, b_i] <- rcat(1, eta)
      B[z[i, b_i], b_i] <- B[z[i, b_i], b_i] + 1
      if (b_i < J) {
        for (j in (b_i + 1):J) {
          jm1 <- j - 1
          z[i, j] <- rcat(1, H[jm1, z[i, jm1], ])
          if (z[i, jm1] < Sp1 & z[i, j] == Sp1) {
            D[z[i, jm1], j] <- D[z[i, jm1], j] + 1
          }
        }
      }

      # observation process
      for (j in b_i:J) {
        if (z[i, j] < Sp1) {
          y[i, j, 1:K[j]] <- rbinom(K[j], 1, p[z[i, j], j]) * z[i, j]

          # multievent
          if (ME) {
            for (k in 1:K[j]) {
              if (y[i, j, k]) {
                y[i, j, k] <- rcat(1, E[z[i, j], ])
              }
            }
          }
        }
      }
    }
  }

  # subset observed individuals and secondaries
```

```
  obs <- which(rowSums(y) > 0)
  I <- length(obs)
  y <- y[obs, , 1:K_max]

  # single state CJS
} else {
  I <- N_super
  f <- sort(sample(1:ifelse(K_max > 1 || ME, J, Jm1), I, replace = T))
  if (S == 1) {
    for (i in 1:I) {
      f_i <- f[i]
      g <- sample(1:K[f_i], 1)
      z[i, f_i] <- y[i, f_i, g] <- 1

      # observation process conditioned on first capture
      for (k in setdiff(1:K[f_i], g)) {
        y[i, f_i, k] <- rbinom(1, 1, p[f_i])
      }
      if (f_i < J) {
        for (j in (f_i + 1):J) {
          jm1 <- j - 1
          z[i, j] <- rbinom(1, 1, z[i, jm1] * phi_tau[jm1])
          y[i, j, 1:K[j]] <- rbinom(1:K[j], 1, z[i, j] * p[j])
        }
      }
    }

    # multistate/multievent
  } else {
    for (i in 1:I) {
      f_i <- f[i]
      g <- sample(1:K[f_i], 1)
      if (ME) {
        z[i, f_i] <- rcat(1, eta)
        y[i, f_i, g] <- rcat(1, E[z[i, f_i], ])
      } else {
        z[i, f_i] <- y[i, f_i, g] <- sample(1:S, 1)
      }
      for (k in setdiff(1:K[f_i], g)) {
        y[i, f_i, k] <- rbinom(1, 1, p[z[i, f_i], f_i]) * z[i, f_i]
        if (ME) {
          if (y[i, f_i, k]) {
```

```
            y[i, f_i, k] <- rcat(1, E[z[i, f_i], ])
          }
        }
      }
      if (f_i < J) {
        for (j in (f_i + 1):J) {
          jm1 <- j - 1
          z[i, j] <- rcat(1, H[jm1, z[i, jm1], ])
          if (z[i, j] < Sp1) {
            y[i, j, 1:K[j]] <- rbinom(K[j], 1, p[z[i, j], j]) * z[i, j]
            if (ME) {
              for (k in 1:K[j]) {
                if (y[i, j, k]) {
                  y[i, j, k] <- rcat(1, E[z[i, j], ])
                }
              }
            }
          }
        }
      }
    }
  }

  # reduce dimensions if single survey
  y <- y[, , 1:K_max]
}


# modify dimensions
if (JS || K_max > 1) {
  p <- p[1:S, ]
} else {
  p <- p[1:S, -1]
}

# initiate output as CJS
variables <- list(h = h, p = p)
generated <- list(I = I, J = J, tau = tau, y = y, ind = ind)

# update robust design
if (K_max > 1) {
  generated <- append(generated, list(K_max = K_max, K = K))
```

```
  }

  # update multistate/multievent
  if (S > 1) {
    generated$S <- S
    variables$q <- q
    if (ME) {
      if (S == 2) {
        variables$`delta[1]` <- delta
      } else {
        variables$delta <- delta
      }
    }
    if (ME || JS) {
      variables$eta <- eta
    }
  }

  # update Jolly-Seber
  if (JS) {
    generated <- append(generated, list(I_aug = I_aug, dirichlet = dirichlet,
                                        collapse = collapse))
    variables <- append(variables,
                        list(mu = mu,
                             gamma = gamma,
                             log_beta = log(beta),
                             N_super = N_super,
                             B = B,
                             D = D,
                             N = apply(z, 2, \(j)
                                       sapply(1:S, \(s) sum(j == s)))))

    # grainsize for CJS within-chain parallelisation
  } else {
    generated$grainsize <- grainsize
  }

  list(variables = variables, generated = generated)
}
```

Utility functions

```r
# transition rate matrix from vectors of mortality and transition rates
rate_matrix <- function(h, q) {
  S <- length(h)
  Sp1 <- S + 1 ; Sm1 = S - 1
  Q <- matrix(0, Sp1, Sp1)
  Q[1:S, Sp1] <- h
  q_s <- head(q, Sm1)
  Q[1, 1] <- -(h[1] + sum(q_s))
  Q[1, 2:S] <- q_s
  if (S > 2) {
    idx <- S
    for (s in 2:Sm1) {
      q_s <- q[idx:(idx + S - 2)]
      Q[s, 1:(s - 1)] <- head(q_s, s - 1)
      Q[s, s] <- -(h[s] + sum(q_s))
      Q[s, (s + 1):S] <- tail(q_s, S - s)
      idx <- idx + Sm1
    }
  }
  q_s <- tail(q, Sm1)
  Q[S, 1:Sm1] <- q_s
  Q[S, S] <- -(h[S] + sum(q_s))
  Q
}

# triangular biadiagonal stochastic matrix
triangular_bidiagonal_stochastic_matrix <- function(delta) {
  E <- diag(c(1, delta))
  for (s in 2:(length(delta) + 1)) {
    sm1 <- s - 1
    E[s, sm1:s] <- c(1 - delta[sm1], delta[sm1])
  }
  E
}

# random categorical draws
rcat <- function(n = 1, prob) {
  rmultinom(n, size = 1, prob) |>
    apply(2, \(x) which(x == 1))
```

```
}

# random Dirichlet draws
rdirch <- function(n = 1, alpha) {
  D <- length(alpha)
  out <- matrix(NA, n, D)
  for (i in 1:n) {
    u <- rgamma(D, alpha, 1)
    out[i, ] <- u / sum(u)
  }
  if (n == 1) {
    out[1, ]
  } else {
    out
  }
}

# plot ECDF-diff and estimates plots together
if (!require(patchwork)) install.packages("patchwork")
plot_sbc <- function(sbc, ..., nrow = NULL, ncol = NULL) {
  patchwork::wrap_plots(
    plot_ecdf_diff(sbc, ...) +
      ggplot2::facet_wrap(~ group,
                          nrow = nrow,
                          ncol = ncol) +
      ggplot2::theme(legend.position = "none"),
    plot_sim_estimated(sbc, ...) +
      ggplot2::facet_wrap(~ variable,
                          nrow = nrow,
                          ncol = ncol,
                          scales = "free"),
    ncol = 1
  )
}
```

References

Gabry, J., R. Češnovar, A. Johnson, and S. Bronder. 2025. cmdstanr: R Interface to 'CmdStan'. Manual.

Hollanders, M., and J. A. Royle. 2022. Know what you don't know: Embracing state uncertainty in disease-structured multistate models. Methods in Ecology and Evolution 13:2827–2837.

Modrák, M., A. H. Moon, S. Kim, P. Bürkner, N. Huurre, K. Faltejsková, A. Gelman, and A. Vehtari. 2023. Simulation-based calibration checking for Bayesian computation: The choice of test quantities shapes sensitivity. Bayesian Analysis -1.

Schwarz, C. J., and A. N. Arnason. 2008. Chapter 12. Jolly-Seber models in MARK. Program MARK: A Gentle Introduction. 19th edition.

Wickham, H., M. Averick, J. Bryan, W. Chang, L. D. McGowan, R. François, G. Grolemund, A. Hayes, L. Henry, J. Hester, M. Kuhn, T. L. Pedersen, E. Miller, S. M. Bache, K. Müller, J. Ooms, D. Robinson, D. P. Seidel, V. Spinu, K. Takahashi, D. Vaughan, C. Wilke, K. Woo, and H. Yutani. 2019. Welcome to the tidyverse. Journal of Open Source Software 4:1686.

Zucchini, W., I. L. MacDonald, and R. Langrock. 2017. Hidden Markov Models for Time Series: An Introduction Using R, Second Edition. Second edition. Chapman and Hall/CRC, New York.