Deterministic SIR Model with Stochastic Cases, with Normalising Flow

```
library(torch)
library(torchflow)
```

SIR model

We first define a prior and data generation function for the SIR model.

For the process model we start with the following two parameters and priors:

- Reproduction number $R_0 \sim \text{Uniform}(1.01, 3)$
- Duration of infection $D \sim \text{Uniform}(2, 6)$

The recovery rate γ and transmission rate β values are then calculated as follows:

- $\gamma = \frac{1}{D}$
- $\beta = R_0 \cdot \gamma$

The population in the simulation is normalised to 1, and we start with 99% of the population susceptible, 1% infected, and 0 recovered:

- $S_1 = 0.99$
- $I_1 = 0.01$
- $R_1 = 0$

The dynamics are, for $t = 2, \ldots, T$::

$$
\begin{aligned}
S_t &= S_{t-1} - \beta S_{t-1} I_{t-1} \\
I_t &= I_{t-1} + \beta S_{t-1} I_{t-1} - \gamma I_{t-1} \\
R_t &= R_{t-1} + \gamma I_{t-1}
\end{aligned}
$$

We run the simulation for $T = 100$ time steps.

For the observation model, we assume an unknown case ascertainment rate (CAR) α and a Poisson distributed number of cases:

- $\alpha \sim \text{Uniform}(0.2, 0.5)$
- $C_t \sim \text{Poisson}(N \cdot \alpha \cdot \beta S_{t-1} I_{t-1})$

where N is the population size (assumed here to be $N = 300$).

The forward simulation code is defined below.

```
generate <- function(
  n,
  R_0_min = 1.01,
  R_0_max = 3,
  D_min = 2,
  D_max = 6,
  alpha_min = 0.2,
```

```
  alpha_max = 0.5,
  N = 300,
  T = 100,
  device = torch_device("cpu")
) {
  R_0 <- torch_rand(n, 1, device = device) * (R_0_max - R_0_min) + R_0_min
  D <- torch_rand(n, 1, device = device) * (D_max - D_min) + D_min
  alpha <- torch_rand(n, 1, device = device) * (alpha_max - alpha_min) + alpha_min
  gamma <- 1 / D
  beta <- R_0 * gamma

  S <- torch_zeros(n, T, device = device)
  I <- torch_zeros(n, T, device = device)
  R <- torch_zeros(n, T, device = device)
  cases <- torch_zeros(n, T, device = device)

  S[, 1] <- 0.99
  I[, 1] <- 0.01
  # Day 1 assumption: all infections are new
  cases[, 1] <- distr_poisson(N * torch_squeeze(alpha) * I[, 1])$sample()

  for (t in 2 : T) {
    new_infections <- torch_squeeze(
      beta * S[, t - 1, drop = FALSE] * I[, t - 1, drop = FALSE]
    )
    new_recoveries <- torch_squeeze(
      gamma * I[, t - 1, drop = FALSE]
    )
    new_cases <- distr_poisson(N * torch_squeeze(alpha) * new_infections)$sample()

    S[, t] <- S[, t - 1] - new_infections
    I[, t] <- I[, t - 1] + new_infections - new_recoveries
    R[, t] <- R[, t - 1] + new_recoveries
    cases[, t] <- new_cases
  }

  list(
    R_0 = R_0,
    D = D,
    gamma = gamma,
    beta = beta,
    alpha = alpha,
    S = S,
    I = I,
    R = R,
    cases = cases,
    N = N
  )
}
```

We generate four example datasets:

```
example_data <- generate(4)
str(example_data)
```

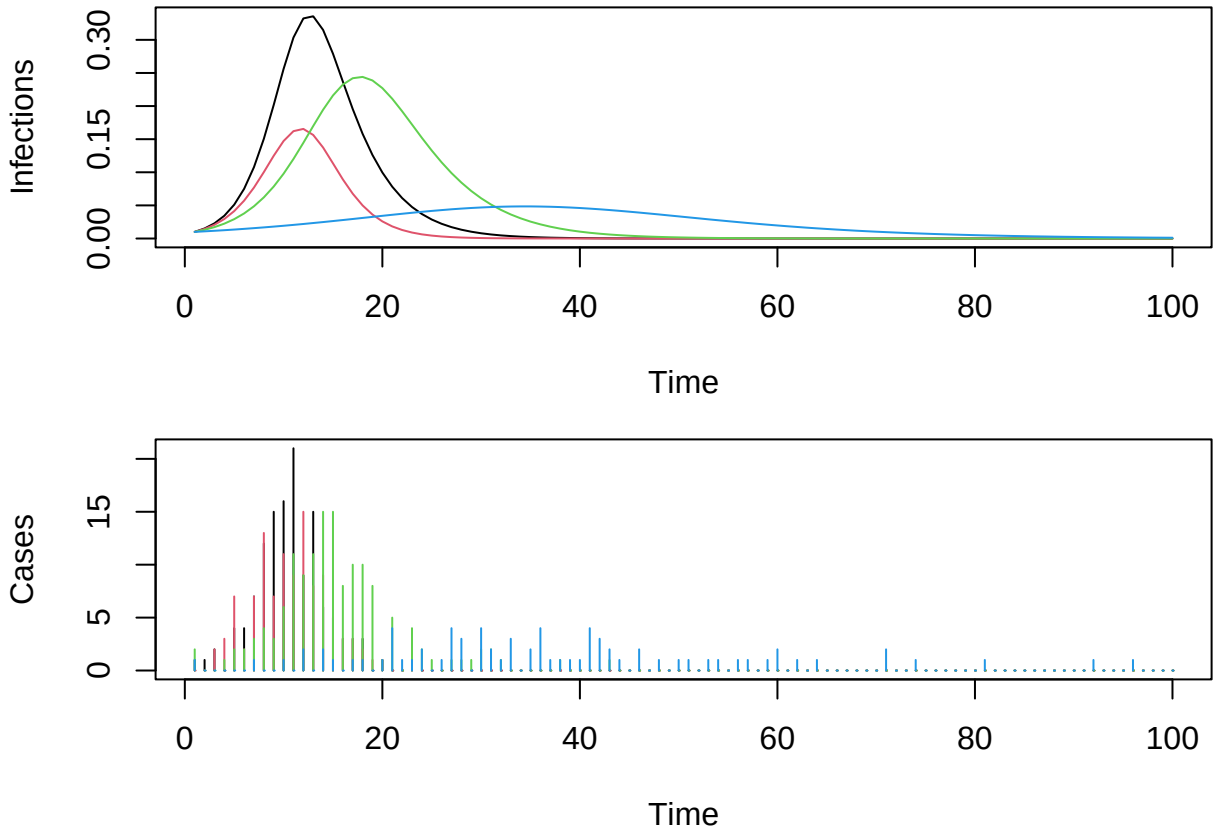
```
## List of 10
##  $ R_0  :Float [1:4, 1:1]
##  $ D    :Float [1:4, 1:1]
##  $ gamma:Float [1:4, 1:1]
##  $ beta :Float [1:4, 1:1]
##  $ alpha:Float [1:4, 1:1]
##  $ S    :Float [1:4, 1:100]
##  $ I    :Float [1:4, 1:100]
##  $ R    :Float [1:4, 1:100]
##  $ cases:Float [1:4, 1:100]
##  $ N    : num 300
```

```
print(example_data[c('gamma', 'beta', 'alpha')])
```

```
## $gamma
## torch_tensor
##  0.2621
##  0.4819
##  0.2221
##  0.2261
## [ CPUFloatType{4,1} ]
##
## $beta
## torch_tensor
##  0.7862
##  0.9346
##  0.5402
##  0.3092
## [ CPUFloatType{4,1} ]
##
## $alpha
## torch_tensor
##  0.4508
##  0.3843
##  0.4472
##  0.4489
## [ CPUFloatType{4,1} ]
```

We can plot the results:

```
withr::with_par(list(mfrow = c(2, 1), mar = c(4, 4, 1, 1)), {
  matplot(t(as_array(example_data$I)), type = "l", lty = 1, xlab = 'Time', ylab = 'Infections')
  matplot(t(as_array(example_data$cases)), type = "h", lty = 1, xlab = 'Time', ylab = 'Cases')
})
```



Normalising flow

We will train a normalising flow to model the posterior distribution $p(\alpha, \beta, \gamma \mid C_1, \ldots, C_{100})$.

A convolutional neural network "summarises" the data by reducing it from a 100-dimensional vector to a 32-dimensional vector.

This is fed into a flow model, which is a sequence of affine coupling blocks and permutation flows.

We train it using generated data and a KL divergence loss function.

First we define the relevant neural network models:

```
# Use GPU if available
device <- if (cuda_is_available()) torch_device("cuda") else torch_device("cpu")

n_summary <- 32L
summary_model <- nn_sequential(
  # Three convolutional layers, with 32, 64, and 128 channels respectively.
  nn_conv1d(1, 32, kernel_size = 5, padding = 2),
  nn_silu(),
  nn_conv1d(32, 64, kernel_size = 5, padding = 2),
  nn_silu(),
  nn_conv1d(64, 128, kernel_size = 5, padding = 2),
  # Take the average of the 128 channels remaining
  nn_adaptive_avg_pool1d(1),
  nn_flatten(),
  # Use an MLP to reduce the 128 channels to 32
```

```
  nn_linear(128, 128),
  nn_silu(),
  nn_linear(128, n_summary)
)

# Define the flow model
n_params <- 3L
flow_model <- nn_sequential_conditional_flow(
  nn_affine_coupling_block(n_params, n_summary),
  nn_permutation_flow(n_params),
  nn_affine_coupling_block(n_params, n_summary),
  nn_permutation_flow(n_params),
  nn_affine_coupling_block(n_params, n_summary)
)

summarizing_flow_model <- nn_summarizing_conditional_flow(summary_model, flow_model)
summarizing_flow_model <- summarizing_flow_model$to(device = device)
```

Then we define a function to generate batches of data for training and testing:

```
generate_conditional_samples <- function(...) {
  n <- 1024L
  data <- generate(n, device = device)
  list(
    target = torch_cat(list(data$alpha, data$beta, data$gamma), dim = -1),
    conditioning = torch_unsqueeze(data$cases, -2)
  )
}
```

We can calculate the initial test loss:

```
test_set <- generate_conditional_samples()
test_loss <- forward_kl_loss(summarizing_flow_model(test_set$target, test_set$conditioning))
cat('Initial test loss:', test_loss$item(), '\n')
```

```
## Initial test loss: 0.3969051
```

Now we train the model for 512 epochs. This uses mini-batches of 256 samples, with total 1024 samples per epoch.

```
system.time(train_conditional_flow(
  summarizing_flow_model,
  generate_conditional_samples,
  n_epochs = 512,
  batch_size = 256,
  after_epoch = function(epoch, ...) {
    if (epoch %% 64 == 0) {
      test_loss <- forward_kl_loss(summarizing_flow_model(test_set$target, test_set$conditioning))
      cat('Epoch', epoch, '| test loss:', test_loss$item(), '\n')
    }
  },
  verbose = interactive()
))
```

```
## Epoch 64 | test loss: -7.942124
## Epoch 128 | test loss: -8.134699
## Epoch 192 | test loss: -8.382236
```

```
## Epoch 256 | test loss: -8.45566
## Epoch 320 | test loss: -8.462072
## Epoch 384 | test loss: -7.64534
## Epoch 448 | test loss: -8.559761
## Epoch 512 | test loss: -8.634985
```

```
##    user  system elapsed
## 117.593   0.075 117.932
```

The training is complete.

Results

Let's generate some samples from the approximate posterior.

We will generate samples from the posterior for the first 8 entries of the test set.

```
example_indices <- 1 : 4
n_samples <- 256
system.time(
  test_samples <- as_array(generate_from_conditional_flow(
    summarizing_flow_model,
    n_samples,
    test_set$conditioning[example_indices, , drop = FALSE]
  ))
)
```

```
##    user  system elapsed
##   0.016   0.002   0.018
```

```
str(test_samples)
```

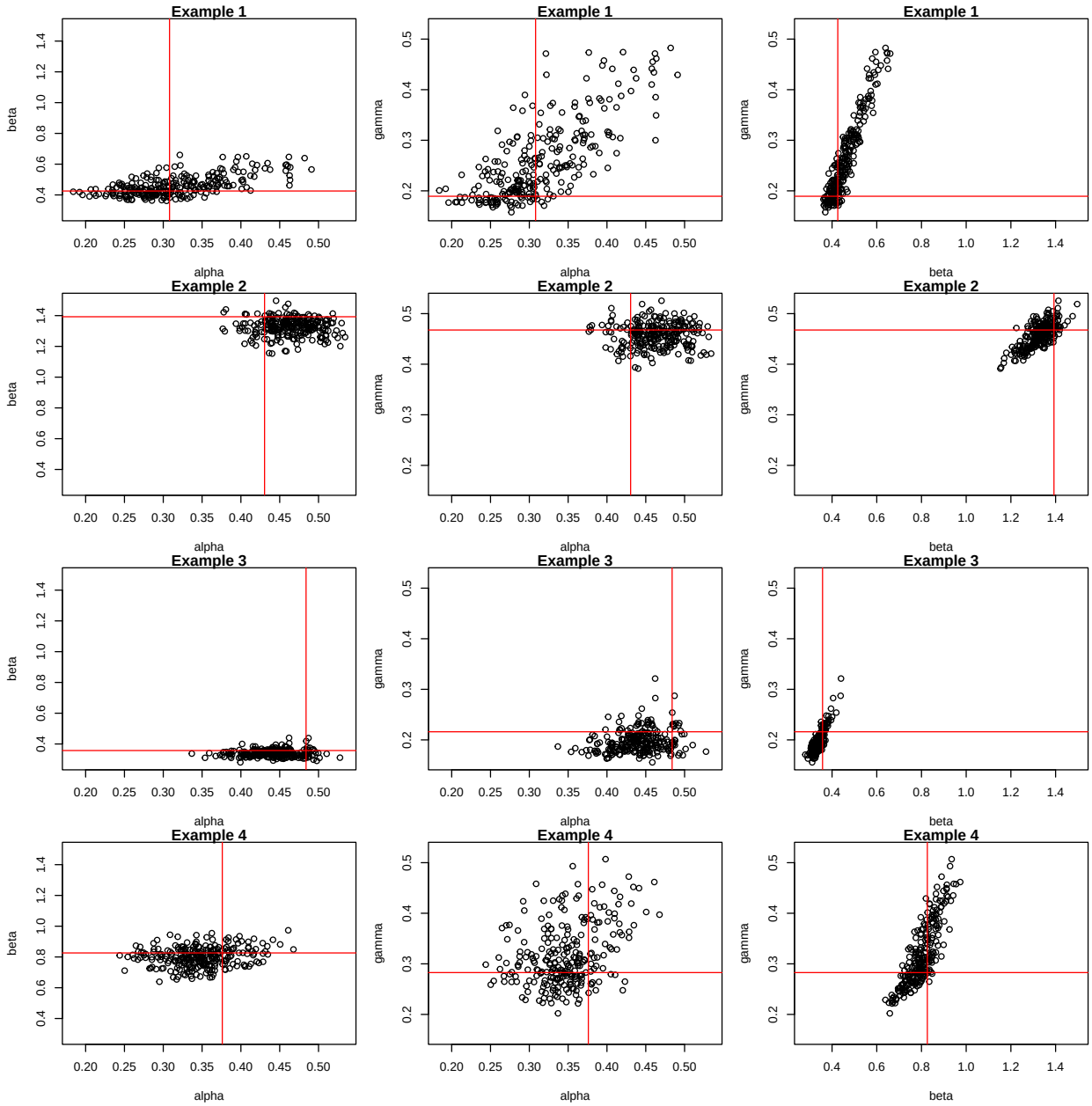
```
##  num [1:256, 1:4, 1:3] 0.235 0.39 0.28 0.37 0.385 ...
```

Now we will plot the samples and the true values:

```
test_target <- as_array(test_set$target)

withr::with_par(list(mfrow = c(4, 3), mar = c(4, 4, 1, 1)), {
  for (i in example_indices) {
    plot_pair <- function(ind1, ind2, lab1, lab2) {
      plot(
        test_samples[, i, ind1],
        test_samples[, i, ind2],
        xlab = lab1,
        ylab = lab2,
        xlim = range(test_samples[, , ind1]),
        ylim = range(test_samples[, , ind2]),
        main = paste('Example', i)
      )
      abline(v = test_target[i, ind1], col = 'red')
      abline(h = test_target[i, ind2], col = 'red')
    }
    plot_pair(1, 2, 'alpha', 'beta')
    plot_pair(1, 3, 'alpha', 'gamma')
    plot_pair(2, 3, 'beta', 'gamma')
  }
```

```
})
```



The samples appear reasonable relative to the true value; more objective comparison would involve using an exact posterior method such as MCMC.

Stan comparison

Now we compare the results to samples from a Stan implementation. First we draw samples:

```
library(cmdstanr)
```

```
## This is cmdstanr version 0.9.0
```

```
## - CmdStanR documentation and vignettes: mc-stan.org/cmdstanr
```

```
## - CmdStan path: /home/mgnb/.cmdstan/cmdstan-2.36.0
## - CmdStan version: 2.36.0
```

```
det_stoch_stan <- cmdstan_model('det-stoch.stan')

test_stan_fits <- lapply(example_indices, function(i) {
  det_stoch_stan$sample(
    data = list(
      N = 300,
      T = 100,
      S_0 = 0.99,
      I_0 = 0.01,
      R_0_min = 1.01,
      R_0_max = 3,
      D_min = 2,
      D_max = 6,
      alpha_min = 0.2,
      alpha_max = 0.5,
      cases = as.integer(as_array(test_set$conditioning[i, 1, ]))
    ),
    seed = 20250709,
    chains = 4,
    refresh = 0,
    parallel_chains = 4
  )
})
```

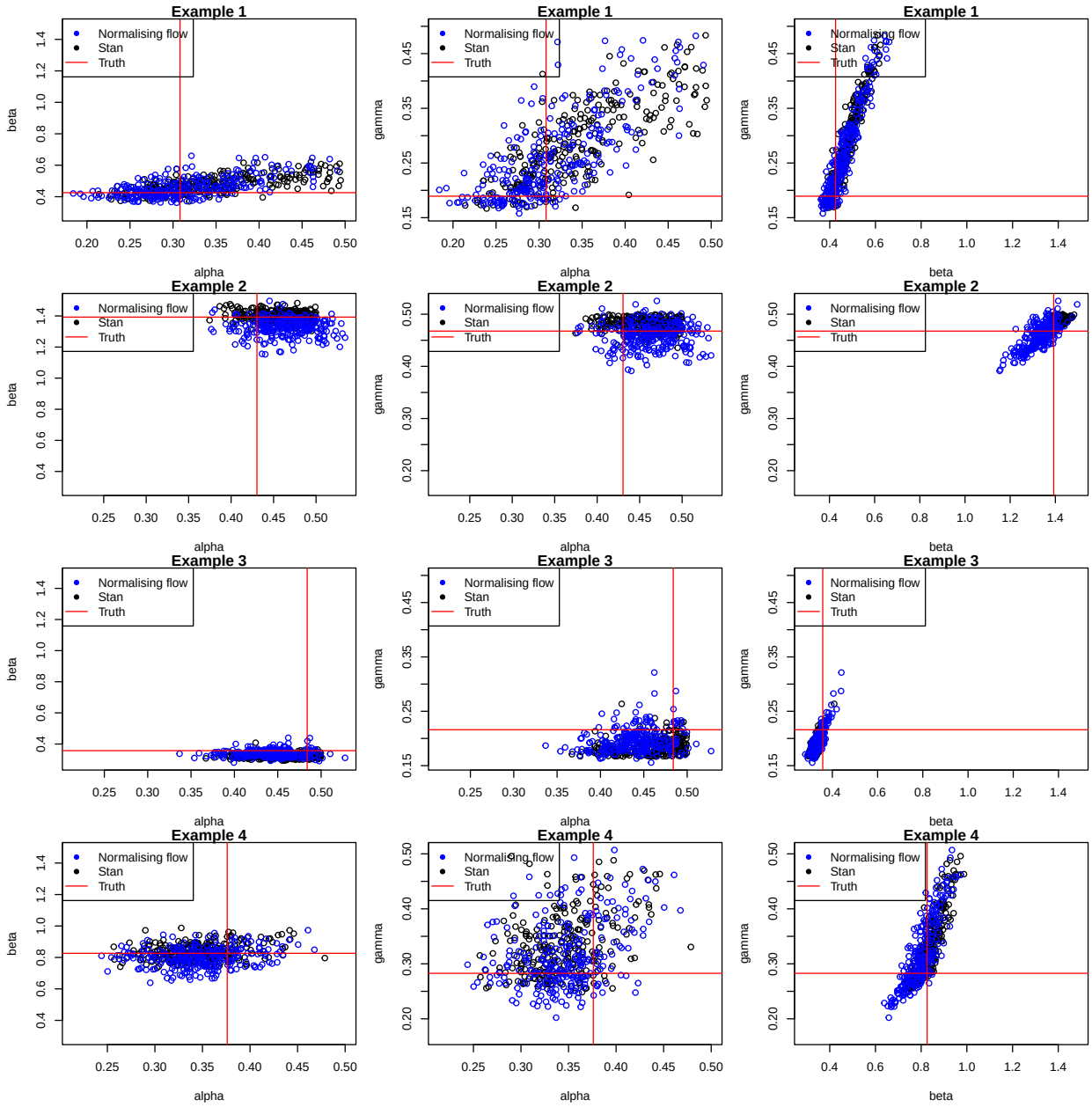
```
## Running MCMC with 4 parallel chains...
##
## Chain 1 finished in 0.2 seconds.
## Chain 2 finished in 0.2 seconds.
## Chain 3 finished in 0.2 seconds.
## Chain 4 finished in 0.2 seconds.
##
## All 4 chains finished successfully.
## Mean chain execution time: 0.2 seconds.
## Total execution time: 0.4 seconds.
##
## Running MCMC with 4 parallel chains...
##
## Chain 1 finished in 0.1 seconds.
## Chain 2 finished in 0.1 seconds.
## Chain 3 finished in 0.1 seconds.
## Chain 4 finished in 0.1 seconds.
##
## All 4 chains finished successfully.
## Mean chain execution time: 0.1 seconds.
## Total execution time: 0.2 seconds.
##
## Running MCMC with 4 parallel chains...
##
## Chain 1 finished in 0.1 seconds.
## Chain 2 finished in 0.1 seconds.
## Chain 3 finished in 0.1 seconds.
```

```
## Chain 4 finished in 0.1 seconds.
##
## All 4 chains finished successfully.
## Mean chain execution time: 0.1 seconds.
## Total execution time: 0.3 seconds.
##
## Running MCMC with 4 parallel chains...
##
## Chain 1 finished in 0.2 seconds.
## Chain 2 finished in 0.2 seconds.
## Chain 3 finished in 0.2 seconds.
## Chain 4 finished in 0.2 seconds.
##
## All 4 chains finished successfully.
## Mean chain execution time: 0.2 seconds.
## Total execution time: 0.3 seconds.
```

Now we produce the same plot as before, but with both Stan and TorchFlow samples":

```r
test_stan_samples <- aperm(abind::abind(lapply(test_stan_fits, function(fit) {
  output <- as.matrix(fit$draws(c('alpha', 'beta', 'gamma'), format = 'draws_matrix'))
  output[floor(seq(1, nrow(output), length.out = n_samples)), ]
}), along = 0), c(2, 1, 3))

withr::with_par(list(mfrow = c(4, 3), mar = c(4, 4, 1, 1)), {
  for (i in seq_along(example_indices)) {
    plot_pair <- function(ind1, ind2, lab1, lab2) {
      plot(
        test_stan_samples[, i, ind1],
        test_stan_samples[, i, ind2],
        xlab = lab1,
        ylab = lab2,
        xlim = range(c(test_stan_samples[, , ind1], test_samples[, i, ind1])),
        ylim = range(c(test_stan_samples[, , ind2], test_samples[, i, ind2])),
        main = paste('Example', i),
      )
      points(test_samples[, i, ind1], test_samples[, i, ind2], col = 'blue')
      abline(v = test_target[example_indices[i], ind1], col = 'red')
      abline(h = test_target[example_indices[i], ind2], col = 'red')
      legend(
        'topleft',
        c('Normalising flow', 'Stan', 'Truth'),
        col = c('blue', 'black', 'red'),
        pch = c(16, 16, NA),
        lty = c(NA, NA, 1)
      )
    }
    plot_pair(1, 2, 'alpha', 'beta')
    plot_pair(1, 3, 'alpha', 'gamma')
    plot_pair(2, 3, 'beta', 'gamma')
  }
})
```


Example 1
Normalising flow
Stan
Truth
beta
alpha
Example 1
gamma
alpha
Example 1
gamma
beta
Example 2
beta
alpha
Example 2
gamma
alpha
Example 2
gamma
beta
Example 3
beta
alpha
Example 3
gamma
alpha
Example 3
gamma
beta
Example 4
beta
alpha
Example 4
gamma
alpha
Example 4
gamma
beta