Simulation-Based Calibration with Stan and RStan

Bob Carpenter

May 2019

Why simulation-based calibration?

Simulation-based calibration (SBC) is a generally applicable method to asses the soundness of an implementation of a Bayesian model and posterior sampler.[1] Well-specified Bayesian models are calibrated by construction. That is, the posterior intervals will have proper frequentist coverage *if the model is correct*.[2] Simulation-based calibration uses this property of Bayesian models to define a testing procedure for Bayesian posterior samplers.

[1] The method was originally developed in Cook, S. R., Gelman, A., & Rubin, D. B. (2006). Validation of software for Bayesian models using posterior quantiles. *Journal of Computational and Graphical Statistics,* 15(3), 675-692. The refinement implemented here is from Talts, S., Betancourt, M., Simpson, D., Vehtari, A., & Gelman, A. (2018). Validating Bayesian inference algorithms with simulation-based calibration. *arXiv* 1804.06788.

[2] Models are typically not correct, which is why we always need to apply posterior predictive checks to assess the fit of real data; simulation-based calibration only assesses the algorithm and model implementation, not its fit to real data.

Bayesian posteriors

Simulation-based calibration relies on the standard application of Bayes's rule and the standard factoring of the joint density. Given a fixed data set y, Bayes's rule tells us that

$$\begin{aligned} p(\theta \mid y) &= \frac{p(y \mid \theta) \cdot p(\theta)}{p(y)} \\ &\propto p(y \mid \theta) \cdot p(\theta) \end{aligned}$$

That is, the posterior $p(\theta \mid y)$ is proportional to the prior $p(\theta)$ times the likelihood $p(y \mid \theta)$.

Simulation-based inference

A sampler provides a sequence of draws distributed according to the posterior distribution,

$$\theta^{(1)}, \ldots, \theta^{(M)} \sim p(\theta \mid y).$$

Samples are useful in that they allow us to calculate integrals corresponding to conditional expectations,

$$\mathbb{E}[f(\theta) \mid y] \;=\; \int f(\theta) \cdot p(\theta \mid y) \, \mathrm{d}\theta.$$

These allow us to compute posterior means as conditional expectations of parameters,

$$\hat{\mu} = \mathbb{E}\left[\mu \mid y\right],$$

event probabilities as conditional expectations of indicator functions,

$$\mathbb{P}[\theta_1 > \theta_2 \mid y] \;=\; \mathbb{E}\left[\mathrm{I}[\theta_1 > \theta_2] \mid y\right],$$

and posterior predictive distributions of new observations as conditional expectations of sampling densities,

$$p(\tilde{y} \mid y) \;=\; \mathbb{E}\left[p(\tilde{y} \mid \theta) \mid y\right].$$

The conditioning in all cases is on observed data y and the expectations are thus taken with respect to the posterior distribution of θ conditioned on y.

Simulation-based Calibration

Simulation-based calibration proceeds by following the generative story of the model and the standard procedure of posterior inference.

Step 1: Simulate from the generative model

The first step is to simulate the value of the parameters according to the prior,

$$\theta^{\text{sim}} \sim p(\theta).$$

Next, it simulates data from the sampling distribution based on the simulated parameter values,[3]

$$y^{\text{sim}} \sim p(y \mid \theta^{\text{sim}}).$$

[3] The distribution $p(y \mid \theta)$ is called the likelihood when considered as a function of the parameters θ for fixed data y, and called the sampling distribution when considered as a function of the variable y for fixed parameters θ.

By construction, $\left(y^{\text{sim}}, \theta^{\text{sim}}\right)$ constitutes a draw from the model's joint density $p(y, \theta)$. By Bayes's rule, we can invert this to show that

$$p\left(\theta^{\text{sim}} \mid y^{\text{sim}}\right) \;\propto\; p\left(y^{\text{sim}}, \theta^{\text{sim}}\right).$$

This is the key insight behind simulation-based calibration—that θ^{sim} is just an ordinary draw from the posterior, $p\left(\theta \mid y^{\text{sim}}\right)$.

Step 2: Sample from the posterior

Next, we use the software being tested to take a sequence of M draws from the posterior given the simulated data,

$$\theta^{(1)}, \ldots, \theta^{(M)} \;\sim\; p\left(\theta \mid y^{\text{sim}}\right).$$

Step 3: Test calibration

From the posterior $p\left(\theta \mid y^{\text{sim}}\right)$ we have a single draw θ^{sim} and a series of draws $\theta^{(1)}, \ldots, \theta^{(M)}$ we would like to test. Because θ^{sim} is a random draw from the posterior just like all of the $\theta^{(m)}$, we know that it should have a uniform distribution in rank when considered among the $\theta^{(1)}, \ldots, \theta^{(M)}$.

This is the hypothesis we are going to test. We will simulate multiple data sets $\left(y^{\text{sim}(n)}, \theta^{\text{sim}(n)}\right)$ for $nin1 : N$ and for each take M draws from the posterior,

$$\theta^{(n,m)} \sim p\left(\theta \mid y^{\text{sim}(n)}\right).$$

For each such simulated data set, we will compute the rank of $\theta^{\text{sim}(n)}$ in $\theta^{(1,m)}, \ldots, \theta^{(N,m)}$,[4]

$$\begin{aligned} r_n &= \text{rank}\left(\theta^{\text{sim}(n)}, \left(\theta^{(n,1)}, \ldots, \theta^{(n,M)}\right)\right) \\ &= 1 + \sum_{m=1}^{M} \text{I}\left[\theta^{(n,m)} < \theta^{\text{sim}(n)}\right] \end{aligned}$$

We then test that the sequence of ranks, $r = r_1, \ldots, r_N$ has a discrete_uniform$(1, M+1)$ distribution. We can do this in any number of ways, but for simplicity, we're going to test it using a simple χ^2 test on binned values.

[4] For example, rank$(4, (5, 1, 6, 7)) = 2$ because there is one numer in the sequence $(5, 1, 6, 7)$ that is less than 4. This operation can also be done by adding the value 4 to the sequence and recovering its usual rank.

Application: Continuous integration testing

Our intended application for these tests is continuous integration testing for Stan.[5] We thus need to automate testing for uniformity.[6]

If θ is multivariate, the entire SBC procedure should be carried out for each component of $\theta = (\theta_1, \ldots, \theta_k)$. To control false positive rates in an automated test framework, we need to adjust the warning thresholds for multiple comparisons.[7]

[5] Continuous integration (CI) tests that nothing breaks when the software is changed.

[6] We can't just follow the advice of Talts et al. and eyeball thousands of histograms every time someone merges a change to Stan!

[7] A more sophisticated approach would be to use a multivariate test for uniformity in order to intrinsically adjust for the multiple comparisons.

Coding Simulation-Based Calibration in Stan

For a given model, we can code up the entire simulation-based calibration procedure in a single Stan program. This includes simulating parameters and data in the transformed data block, defining the posterior to be sampled in the parameters and model blocks, and then defining the indicator function $\text{I}[\theta^{(m)} < \theta^{\text{sim}}$ as a generated quantity. Sampling from the posterior of this model in Stan allows us to compute ranks by summing the indicator over the M draws.

The model

We will be testing a very simple model that estimates a normal location parameter $\mu \in (-\infty, \infty)$ and scale parameter $\sigma \in (0, \infty)$.[8] We will use the independent priors

$$\mu \sim \text{normal}(0, 1)$$

and

$$\sigma \sim \text{lognormal}(0, 1).$$

[8] Thus we have $\theta = (\mu, \sigma)$ in the notation of the previous section.

The prior density is thus

$$p(\mu, \sigma) \;=\; \text{normal}(\mu \mid 0, 1) \cdot \text{lognormal}(\sigma \mid 0, 1).$$

We assume that the data consists of a vector $y = y_1, \ldots, y_{10}$, each element of which is generated independently according to

$$y_n \sim \text{normal}(\mu, \sigma).$$

Thus our likelihood is

$$p(y \mid \mu, \sigma) \;=\; \prod_{n=1}^{10} \text{normal}(y_n \mid \mu, \sigma).$$

Stan program

Here's the complete Stan program implementing the model defined in the previous section.

```
transformed data {
  real mu_sim = normal_rng(0, 1);
  real<lower = 0> sigma_sim = lognormal_rng(0, 1);

  int<lower = 0> N = 10;
  vector[N] y_sim;
  for (n in 1:N)
    y_sim[n] = normal_rng(mu_sim, sigma_sim);
}
parameters {
  real mu;
  real<lower = 0> sigma;
}
model {
  mu ~ normal(0, 1);
  sigma ~ lognormal(0, 1);

  y_sim ~ normal(mu, sigma);
}
generated quantities {
  int<lower = 0, upper = 1> I_lt_sim[2]
      = { mu < mu_sim, sigma < sigma_sim };
}
```

Let's walk through the program line by line. First, there is no `data` block, so no external data needs to be provided to the program to run. It defines constant size variables (`N`) by assignment and uses random number generators to define the simulated parameters (`mu_sim`, `sigma_sim`), and the simulated data (`y_sim`).

The first statement draws a random value for `mu_sim` from a standard normal distribution using Stan's random number generation capabilities. The second statement draws a positive random variable from a standard lognormal distribution. This provides our values for our simulated parameters,

$$\theta^{\text{sim}} \ = \ \left(\mu^{\text{sim}}, \sigma^{\text{sim}}\right).$$

Next, we declare a non-negative integer variable `N` and define it to have the value 10. That will determine the size of the simulated data vector

$$y^{\text{sim}} \ = \ \left(y_1^{\text{sim}}, \ldots, y_N^{\text{sim}}\right).$$

Finally, we have a loop to generate the y_n^{sim} values independently according to a normal distribution with location μ^{sim} and scale σ^{sim}.

Next up, we declare the two scalar parameters, `mu` and `sigma`, for the model we are going to fit in the `parameters` block. The variable `sigma` must be defined with the lower bound of zero, as it is required to be positive.[9]

[9] If it were not constrained, Stan would try to explore negative values for it. When values are constrained to be positive, Stan will transform the geometry of the space it explores in order to restrict its attention to positive values.

The first two sampling statements in the `model` block define the prior, $p(\mu, \sigma) = p(\mu) \cdot p(\sigma)$. The last statement defines the likelihood, $p(y^{\text{sim}} \mid \mu, \sigma)$. This means that Stan's going to draw posterior samples according to

$$\left(\mu^{(1)}, \sigma^{(1)}\right), \ldots \left(\mu^{(M)}, \sigma^{(M)}\right) \sim p\left(\mu, \sigma \mid y^{\text{sim}}\right),$$

which is what we need for simulation-based calibration. The sample is based on the posterior for the simulated data y^{sim}.

Finally, the `generated quantities` block declares and defines an integer array `I_lt_sim` of boolean values. The name follows the definition of applying the indicator function to the test that the simulated data was less than a simulated parameter value. The test in our example program returns an array with two values, the first entry of which is an indicator of whether $\mu < \mu^{\text{sim}}$ and the second and indicator of whether $\sigma < \sigma^{\text{sim}}$.[10]

[10] We conventionally order these in the same order as the parameters were declared to make it easy to read them out in order downstream.

In general, if we have a posterior draw $\theta^{(m)}$, the value for k-th entry of $\text{I\_lt\_sim}^{(m)}$ will be

$$\text{I\_lt\_sim}_k^{(m)} \ = \ \text{I}\left[\theta_k^{(m)} < \theta_k^{\text{sim}}\right].$$

In our example program, let's suppose our simulated parameters turn out to be

$$(\mu^{\text{sim}}, \sigma^{\text{sim}}) \ = \ (1.01, 0.23).$$

Suppose we then take $M = 4$ simulations to produce draws

m	$\mu^{(m)}$	$\sigma^{(m)}$
1	1.07	0.33
2	−0.32	0.14
3	−0.99	0.26
4	1.51	0.31

Then the value of I_lt_sim will be an array with four rows and two columns,

m	$\mu^{(m)} < \mu^{\text{sim}}$	$\sigma^{(m)} < \sigma^{\text{sim}}$
1	0	0
2	1	1
3	1	0
4	0	0

In two of the three posterior draws, $\mu^{(m)} < \mu^{\text{sim}}$; in one of the posterior draws, $\sigma^{(m)} < \sigma^{\text{sim}}$. The rank of μ^{sim} and σ^{sim} are thus the sums of the columns plus one (to ensure ranks run from 1 to $M + 1$).

The number of possible ranks for the simulated parameter among the posterior draws is one larger than the number of posterior draws. For example, if we have five posterior draws $\mu^{(1)}, \ldots, \mu^{(5)}$ for the location parameter, the rank of the simulated parameter μ^{sim} can be as low as 0 if it's smaller than all of the $\mu^{(m)}$ and as high as 5 if it's larger than all of the $\mu^{(m)}$.[11] We add one to the raw counts (which range from zero the number of draws) to get ranks numbering from one to the number of draws plus one.

[11] This is called the "fencepost" problem in introductory computer science classes, where the analogy is that if there are n fence posts, there are $n - 1$ connecting bits of fence. Failure to adjust for this discrepancy in counts between posts and fence units is the cause of numerous off-by-one errors in computer programs.

Hypothesis Testing Uniformity of Ranks

We're going to apply a simple χ^2 test to the ranks by binning them. We will divide the ranks into 20 bins.[12]

Because the number of ranks is one greater than the number of simulations, we will use 999 simulations to get an evenly divisible 1000 possible ranks. Also, to make our life easier in R, we will number the ranks `1:1000` rather than `0:999`.

[12] This assumes the number of ranks is divisible by 20 to ensure uniformity; converting to floating point does not solve this problem, as we only have a discrete number of possible ranks as output.

The counts in each bin will follow a uniform distribution under the null hypothesis of calibration. Therefore the squared difference from the expected count of each bin will follow a χ^2 distribution and we can employ the standard hypothesis test for uniformity. This involves the test statistic

$$X^2 = \sum_{i=1}^{I} \frac{(b_i - e_i)^2}{e_i}$$

where b_i is the number of ranks falling in bin $i \in 1 : I$ and

$$e_i = \frac{M}{I}$$

is the number of ranks expected to fall in that bin under the null hypothesis of uniformity.[13] Under the null hypothesis of uniformity, the test statistic X^2 follows a χ^2 distribution with I degrees of freedom.[14] The reported p-value will be that of the probability of having a value as extreme as that observed in the b_i given the assumption of uniformity. If this is very small, we can confidently reject the assumption of uniformity and conclude there is something wrong with our sampler.[15]

[13] If the bins are not uniformly sized, the expectations e_i may be changed to accomodate.

[14] The test statistic is just a function of the random variables b_i and is hence a random variable itself with all the usual properties like having a distribution.

[15] We do not need to make such stark binary choices. We are just using these hypothesis tests as flags to alert us to possible problems introduced by code changes or to cause us to check our model's fit more closely if we are just applying this procedure to a single model of interest.

Testing uniformity in R

To implement our uniformity test in R, we use

```
# @param y:  sequence of ranks in 1:max_rank
# @param max_rank: maximum rank of data in y
# @param bins (default 20):  bins to use for chi-square test
# @error return NA if max rank not divisible by number of bins
# @return p-value for chi-square test that data is evenly
#         distributed among the bins
test_uniform_ranks <- function(y, max_rank, bins = 20) {
  if (max_rank / bins != floor(max_rank / bins)) {
     printf("ERROR in test_uniform_ranks")
     printf("  max rank must be divisible by bins.")
     printf("  found max rank = %d;  bins = %d", max_rank, bins)
     return(NA)
  }
  bin_size <- max_rank / bins
  bin_count <- rep(0, bins)
  N <- length(y)
  for (n in 1:N) {
    bin <- ceiling(y[n] / bin_size)
    bin_count[bin] <- bin_count[bin] + 1
  }
  chisq.test(bin_count)$p.value
}
```

As input, this function takes the sequence `y` of ranks to evaluate for uniformity, the maximum possible rank, and the number of bins to use (defaulting to 20). After checking consistency, it just iterates through the elements of `y` incrementing the appropriate bin. This can be done with a simple rounded integer division as shown. If the number of ranks is divisible by the number of bins, we expect

each bin to have the same number of elements. Finally, the built-in `chisq.test` of R is used and its p-value returned.[16]

[16] We could refine this function to deal with uneven bin sizes by adding a parameter to the `chisq.test` call that indicates the expected probability of inclusion in each bin.

Coding Simulation-Based Calibration in RStan

Given the Stan program to do the heavy lifting of fitting, we can write a simple R driver program to calculate all we need for simulation-based calibration (assuming a Stan program defining the appropriate test variables). Let's first include the Stan library and print.

First, we have a simple program to determine the number of parameters being monitored for simulation-based calibration in the Stan program. This and the SBC program itself will depend on the `rstan` library.

```
# @param model: precompiled Stan model
# @param data: data for model (defaults to empty list)
# @return size of the generated quantity array I_lt_sim
num_monitored_params <- function(model, data = list()) {
    fit <- sampling(model, data = data,
                    iter = 1, chains = 1, warmup = 0,
                    refresh = 0, seed = 1234)
    fit@par_dims$I_lt_sim
  }
```

The SBC program itself takes a slew of arguments and returns a structured result.

```
# @param model: precompiled Stan model
# @param data: list of data for model (defaults to empty)
# @param sbc_sims:  number of total simulation runs for SBC
# @param stan_sims: number of posterior draws per Stan simulation
# @param init_thin: initial thinning (doubles thereafter up to max)
# @param max_thin: max thinning level
# @param seed: PRNG seed to use for Stan program to generate data
# @param target_n_eff: target effective sample size (should be 80%
#                      or 90% of stan_sims to stand a chance)
# @return list with keys (rank, p_value, thin) for 2D array of ranks
#         and 1D array of p-values, and 1D array of thinning rates
sbc <- function(model, data = list(),
                sbc_sims = 1000, stan_sims = 999,
                init_thin = 4, max_thin = 64,
                target_n_eff = 0.8 * stan_sims) {
  num_params <- num_monitored_params(model, data)
  ranks <- matrix(nrow = sbc_sims, ncol = num_params)
  thins <- rep(NA, sbc_sims)
```

```
  for (n in 1:sbc_sims) {
    n_eff <- 0
    thin <- init_thin
    while (TRUE) {
      fit <- sampling(model,
                      data = data,
                      chains = 1,
                      iter = 2 * thin * stan_sims,
                      thin = thin,
                      control = list(adapt_delta = 0.99),
                      refresh = 0)
      fit_summary <- summary(fit, pars = c("lp__"), probs = c())$summary
      n_eff <- fit_summary["lp__", "n_eff"]
      if (n_eff >= target_n_eff || (2 * thin) > max_thin) break;
      thin <- 2 * thin
    }
    thins[n] <- thin
    # printf("n = %5d;  thin = %4d;  n_eff = %5.0f", n, thin, n_eff)
    lt_sim <- extract(fit)$I_lt_sim
    for (i in 1:num_params)
      ranks[n, i] <- sum(lt_sim[ , i]) + 1
  }
  pval <- rep(NA, num_params)
  for (i in 1:num_params)
    pval[i] <- test_uniform_ranks(ranks[ , i],
                                  max_rank = stan_sims + 1)
  list(rank = ranks, p_value = pval, thin = thins)
}
```

The arguments to the `sbc()` function include a precompiled Stan model[17], any data required by that model (defaulting to an empty list for models that require no external data), a number of total simulations for SBC and a total number of posterior draws per simulation for Stan. Additional arguments control the initial thinning rate, the maximum thinning rate, and the target effective sample size. The thinning rate will start at the initial value and be increased until we get at least the target effective sample size or exceed the maximum thinning rate.

[17] As produced from a Stan program by `rstan::stan_model(model_file)`.

Before doing anything else, the program calls `num_monitored_params` to find out the size of the the indicator array so it can preallocate the matrix of ranks produced by sampling.

The `sbc()` function then iterates over the number of SBC simulations, and for each one simulates a data set then fits it with Stan.[18] We need to thin and make sure the resulting effective sample size is

[18] The simulation code for the parameters is also in the Stan program.

large enough to remove correlation from the posterior draws or they will fail the uniformity test.[19]

In order to hit our target effective sample size, we'll introduce a technique known as *iterative deepening* in the algorithms literature. The algorithm starts runs with an initial amount of thinning, then tests if the effective sample size is large enough. If it's not, the program continues retrying with more iterations and more thinning until it hits the target effective size or exceeds the maximum thinning rate allowed.[20]

To monitor mixing, we will be using the estimated effective sample size for `lp__`, the log density (up to an additive normalizing constant). This is a non-linear function of all of the parameters and if it mixes well, the parameters typically mix well.[21] A stricter test would be to require every parameter's estimated effective sample size to exceed some threshold; multiple parameters will make such a test even stricter.

The call to the sampler uses the seed specified in the argument, with a chain id corresponding to the SBC iteration. This will make sure each SBC iteration uses its own segment of a long sequence of pseudorandom number draws.[22]

To extract the effective sample size for `lp__`, we need to dive into the returned Stan fit object.[23] We extract the number of effective samples for the variable `lp__`, which represents the log density of the model at the sample being drawn (up to an additive normalizing constant).

If the effective size is large enough, or if the maximum thinning rate will be exceeded in the next iteration, the algorithm breaks out of the loop.[24] Otherwise, the algorithm doubles the thinning rate and tries again. The additional factor of two in the number of iterations is because half of the iterations go to warmup by default. As it goes along, the program records the thinning rate in each iteration.

When it has found a large enough estimated effective sample size for the SBC iteration (or fails by exceeding max thinning), it then extracts the value of the generated quantity `I_lt_sim`. The value of `I_lt_sim[m, k]` is a binary indicator for posterior draw `m` and parameter `k` as to whether the posterior draw's value is less than the simulated value for the parameter.

Then, for each parameter `k`, we sum the indicators `I_lt_sim[ , k]` to calculate the rank of the each simulated parameter among the posterior draws for that parameter. Because we add one, the value will be between one and one plus the maximum number of Stan draws. With a default of 999 draws, the ranks should range uniformly from 1 to 1000.

With these ranks in hand, the `test_uniform_ranks` program calcu-

[19] This doesn't mean we should do inference on thinned sets of draws; the unthinned draws produce more accurate expectation calculations.

[20] If there was no maximum, there would be a risk of infinite loops in models that do not mix well in the sampler.

[21] As the transform of a group of random variables, the log density is also a random variable in the posterior. We could use it to compute the entropy of a distribution, for example, which is the expectation of a log density.

[22] Controlling the pseudo-random number generators in simulation programs is critical. Often built-in seeds are time-based, which in computation-intensive situations can result in the same seed being reused, which defeats the assumption of independent testing.

[23] RStan is distributed with a vignette on the Stan fit object. R uses the term "vignette" for documentation of how to use a package.

[24] The `break` statement causes execution of a loop to terminate and execution to begin after the loop; it's useful when there are complicated termination conditions that are awkward to write in the loop's condition.

lates the p-value for each parameter. The null hypothesis is that the ranks of the simulated parameter value among the posterior draws will be uniform across multiple simulated parameters and data sets.

The program returns the raw ranks for each parameter for each SBC iteration, along with the p-values for each parameter, and the thinning level used for each iteration.

When things go right

To run, we first have to compile the Stan program.

```
model <- stan_model("normal-sbc.stan")
```

Then we can call the simulation-based calibration function.

```
result <- sbc(model, data = list(), sbc_sims = 1000, stan_sims = 999,
              max_thin = 64)
```

With 999 Stan simulations, there are 1000 possible ranks, so the bins will divide nice and evenly. We can see the thinning levels actually used by summarizing them as a table.

```
table(result$thin)
```

```
  4   8  16  32
563 424  12   1
```

Finally, we print out the p-values for our uniformity test.

```
result$p_value
```

```
 [1] 0.77 0.75
```

Everything looks good.[25]

We can also plot histograms of the ranks.

[25] We'd be worried if those p-values were very small.

```
A <- dim(result$rank)[1]
rank_df <-
  rbind(data.frame(parameter = rep("mu", A),
                   y = result$rank[ , 1]),
        data.frame(parameter = rep("sigma", A),
                   y = result$rank[, 2]))
rank_plot <-
  ggplot(rank_df, aes(x = y)) +
  geom_histogram(binwidth = 50, color = "black",
                 fill = "#ffffe8", boundary = 0) +
  facet_wrap(vars(parameter)) +
```

```
  ggtheme_tufte() +
  theme(panel.spacing.x = unit(2, "lines"))
rank_plot
```

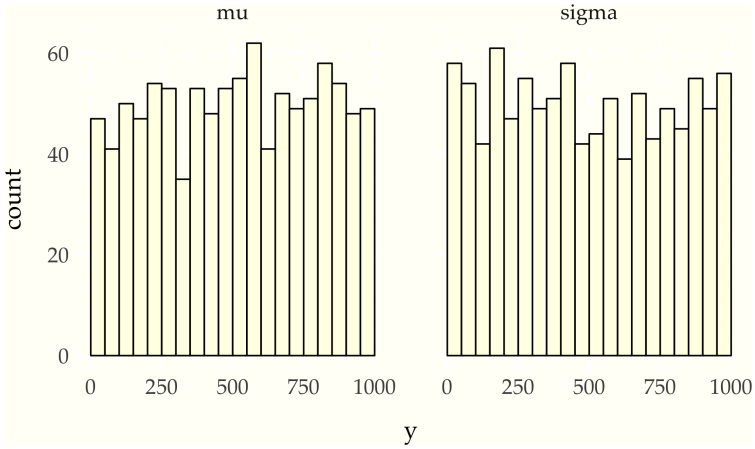

Figure 1: Histogram of ranks of the simulated parameter value with respect to the posterior draws for the two model parameters. If all is working as it should be, these should look uniform, which they do here.

When things go wrong

Now let's see what happens when our model is misspecified for our data generating process. To set this up, we'll use the same normal model, but rather than generating normal data, we'll generate Student-t-distributed data with four degrees of freedom. We'll simulate μ^{sim} and σ^{sim} as before (standard normal and standard lognormal) and generate the simulated y^{sim} based on these as

$$y_n^{\text{sim}} \sim \text{student\_t}(4, \mu, \sigma).$$

Here's the full Stan code.

```
print_file('bad-t-normal-sbc.stan')
  transformed data {
    real mu_sim = normal_rng(0, 1);
    real<lower = 0> sigma_sim = lognormal_rng(0, 1);

    int<lower = 0> N = 10;
    vector[N] y_sim;
    for (n in 1:N)
      y_sim[n] = student_t_rng(4, mu_sim, sigma_sim);
  }
  parameters {
    real mu;
    real<lower = 0> sigma;
  }
```

```
model {
  mu ~ normal(0, 1);
  sigma ~ lognormal(0, 1);

  y_sim ~ normal(mu, sigma);
}
generated quantities {
  int<lower = 0, upper = 1> I_lt_sim[2]
      = { mu < mu_sim, sigma < sigma_sim };
}
```

Only a single line has changed in the code, with

```
y_sim[n] = normal_rng(mu_sim, sigma_sim);
```

being replaced by

```
y_sim[n] = student_t_rng(4, mu_sim, sigma_sim);
```

Now let's see what happens. We need to compile the model, run SBC, and print the p-values.

```
bad_model <- stan_model('bad-t-normal-sbc.stan')
bad_result <- sbc(bad_model, data = list(),
                  sbc_sims = 1000, stan_sims = 999,
                  max_thin = 64)

bad_result$p_value
  [1] 0.536 0.000
```

The resulting p-values show a clear failure of calibration, as we would expect. Here are the histograms of ranks, which visually indicate how the location parameter is well calibrated but not the scale parameter.

```
bad_A <- dim(bad_result$rank)[1]
bad_rank_df <-
  rbind(data.frame(parameter = rep("mu", bad_A),
                   y = bad_result$rank[ , 1]),
        data.frame(parameter = rep("sigma", bad_A),
                   y = bad_result$rank[, 2]))
bad_rank_plot <-
  ggplot(bad_rank_df, aes(x = y)) +
  geom_histogram(binwidth = 50, color = "black",
                 fill = "#ffffe8", boundary = 0) +
  facet_wrap(vars(parameter)) +
```

```
  ggtheme_tufte() +
  theme(panel.spacing.x = unit(2, "lines"))
bad_rank_plot
```

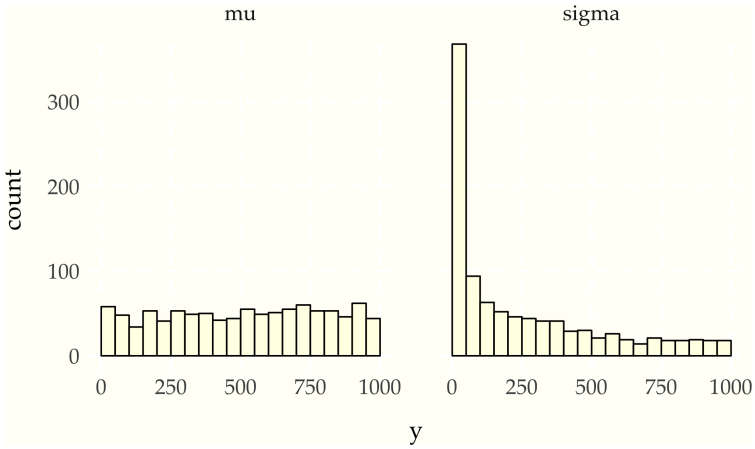

Figure 2: For a misspecified model where the simulation procedure does not match the model, the histograms will be far uniform. Here, the data generating process is Student-t with four degrees of freedom whereas the model assumes normality.

We see the problem immediately in that we are way off in estimating σ in the posterior; which shows up directly in the small p-value.

Conclusion

When we generate data from the true generating process of a model, we expect our sampling program to be able to sample from the posterior given the data. Simulation-based calibration lets us test if our samplers are properly sampling from the posterior of a given model by testing simulated coverage. This note shows how to code simulation-based calibration for a Stan model using an elaborated Stan model and how to drive such tests using RStan.

Acknowledgements

Thanks to Lauren Kennedy for helping me with R data structures[26]. And thanks to Jonah Gabry for the trick on using an integer array of comparison points to make the code more general; he already had SBC implemented for Bayesplot on a branch before I wrote this case study. Please don't blame Aki Vehtari for my poor uniformity tests—he gave me advice on better ones I just haven't acted on yet.

[26] If you don't know `str()`, you should. It's like magic. I can finally use a Stan fit object without the vignette.

Appendix: Included R code

This all executes before other code, but is not included above to cluttering the document before it starts. We need to import the following R libraries.

```
library(ggplot2); library(knitr); library(rstan);  library(tufte)
```

We also have some general print utility functions.

```
println <- function(msg) cat(msg); cat("\n")
printf <- function(pattern, ...) println(sprintf(pattern, ...))
print_file <- function(file) println(readLines(file))
```

We also have some general R configuration.

```
options(digits = 2);  options(htmltools.dir.version = FALSE)
```

There's also configuration for knitr itself.

```
knitr::opts_chunk$set(
  include = TRUE,  cache = FALSE,  collapse = TRUE,  echo = TRUE,
  message = FALSE, tidy = FALSE,  warning = FALSE,   comment = "  ",
  dev = "png",  dev.args = list(bg = '#FFFFF8'),  dpi = 300,
  fig.align = "center",  fig.width = 7,  fig.asp = 0.618,  fig.show = "hold",
  out.width = "90%")
```

And finally, configuration for the theme we use to match the Tufte handout format for R markdown.

```
ggtheme_tufte <- function() {
  theme(plot.background =
          element_rect(fill = "#fffff8",
                       colour = "#fffff8",
                       size = 0.5,
                       linetype = "solid"),
        plot.margin=unit(c(1, 1, 0.5, 0.5), "lines"),
        panel.background =
          element_rect(fill = "#fffff8",
                       colour = "#fffff8",
                       size = 0.5,
                       linetype = "solid"),
        panel.grid.major = element_line(colour = "white",
                                        size = 1, linetype="dashed"),
        panel.grid.minor = element_blank(),
        legend.box.background =
          element_rect(fill = "#fffff8",
                       colour = "#fffff8",
                       linetype = "solid"),
        axis.ticks = element_blank(),
        axis.text = element_text(family = "Palatino", size = 14),
        axis.title.x = element_text(family = "Palatino", size = 16,
                                    margin = margin(t = 15,
```

```
                                                r = 0, b = 0, l = 0)),
        axis.title.y = element_text(family = "Palatino", size = 16,
                                    margin = margin(t = 0,
                                                    r = 15, b = 0, l = 0)),
        strip.background = element_rect(fill = "#fffff8",
                                        colour = "#fffff8",
                                        linetype = "solid"),
        strip.text = element_text(family = "Palatino", size = 14),
        legend.text = element_text(family = "Palatino", size = 14),
        legend.title = element_text(family = "Palatino", size = 16,
                                    margin = margin(b = 5)),
        legend.background = element_rect(fill = "#fffff8",
                                        colour = "#fffff8",
                                        linetype = "solid"),
        legend.key = element_rect(fill = "#fffff8",
                                        colour = "#fffff8",
                                        linetype = "solid")
  )
}
```

Appendix: Session information

In the interest of full reproducibility, here is a record of the dependencies used to generate this case study.

```
sessionInfo()
   R version 3.5.0 (2018-04-23)
   Platform: x86_64-apple-darwin15.6.0 (64-bit)
   Running under: macOS High Sierra 10.13.6

   Matrix products: default
   BLAS: /Library/Frameworks/R.framework/Versions/3.5/Resources/lib/libRblas.0.dylib
   LAPACK: /Library/Frameworks/R.framework/Versions/3.5/Resources/lib/libRlapack.dylib

   locale:
   [1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8

   attached base packages:
   [1] stats     graphics  grDevices utils
   [5] datasets  methods   base

   other attached packages:
   [1] tufte_0.4          rstan_2.18.1
   [3] StanHeaders_2.18.0 knitr_1.20
   [5] ggplot2_3.1.0      rmarkdown_1.10
```

```
loaded via a namespace (and not attached):
 [1] Rcpp_0.12.18       highr_0.6
 [3] pillar_1.2.3       compiler_3.5.0
 [5] plyr_1.8.4         bindr_0.1.1
 [7] prettyunits_1.0.2  base64enc_0.1-3
 [9] tools_3.5.0        digest_0.6.15
[11] pkgbuild_1.0.2     evaluate_0.10.1
[13] tibble_1.4.2       gtable_0.2.0
[15] pkgconfig_2.0.2    rlang_0.2.1
[17] cli_1.0.0          parallel_3.5.0
[19] yaml_2.2.0         xfun_0.1
[21] loo_2.0.0          bindrcpp_0.2.2
[23] gridExtra_2.3      withr_2.1.2
[25] dplyr_0.7.6        stringr_1.3.1
[27] stats4_3.5.0       rprojroot_1.3-2
[29] grid_3.5.0         tidyselect_0.2.4
[31] glue_1.2.0         inline_0.3.15
[33] R6_2.2.2           processx_3.2.0
[35] callr_3.0.0        purrr_0.2.5
[37] magrittr_1.5       codetools_0.2-15
[39] matrixStats_0.54.0 ps_1.2.0
[41] backports_1.1.2    scales_0.5.0
[43] htmltools_0.3.6    assertthat_0.2.0
[45] colorspace_1.3-2   labeling_0.3
[47] tinytex_0.5        stringi_1.2.2
[49] lazyeval_0.2.1     munsell_0.4.3
[51] crayon_1.3.4
```