Markov Chain Monte Carlo Diagnostics

Michael Betancourt

10/1/22

Table of contents

In this short note I will preview the new suite of Markov chain Monte Carlo analysis tools that I will be introducing more formally in upcoming writing. These tools largely focus on diagnostics but there are also a few that cover Markov chain Monte Carlo estimation assuming a central limit theorem.

We'll start with diagnostics specific to Hamiltonian Monte Carlo then consider more generic diagnostics that consider each expectand of interest one at a time. Finally we'll look at a way to visualize one-dimensional pushforward distributions using Markov chain Monte Carlo to estimate bin probabilities.

Before any of that, however, we need to set up our graphics.

```r
par(family="serif", las=1, bty="l", cex.axis=1, cex.lab=1, cex.main=1,
    xaxs="i", yaxs="i", mar = c(5, 5, 3, 5))

c_light <- c("#DCBCBC")
c_light_highlight <- c("#C79999")
c_mid <- c("#B97C7C")
c_mid_highlight <- c("#A25050")
c_dark <- c("#8F2727")
c_dark_highlight <- c("#7C0000")
```

1 Hamiltonian Monte Carlo Diagnostics

Hamiltonian Monte Carlo introduces a suite of powerful diagnostics that can identify obstructions to Markov chain Monte Carlo central limit theorems. These diagnostics are not only extremely sensitive but also probe the behavior of the entire Markov chain state instead of the projections of that state through single expectands.

1.1 Check Hamiltonian Monte Carlo Diagnostics

All of our diagnostics are assembled in this single `check_all_hmc_diagnostics` function.

The first diagnostic looks for unstable numerical Hamiltonian trajectories, or divergences. These unstable trajectories are known to obstruct typical central limit theorem conditions. Divergences arise when the target distribution is compressed into a narrow region; this forces the Hamiltonian dynamics to accelerate which makes them more difficult to accurately simulate.

Increasing `adapt_delta` will on average result in a less aggressive step size optimization that in some cases may improve the stability of the numerical integration but at the cost of longer, and hence more expensive, numerical Hamiltonian trajectories. In most cases, however, the only productive way to avoid divergences is to reparameterize the ambient space to decompress these pinches in the target distribution.

Stan's Hamiltonian Monte Carlo sampler expands the length of the numerical Hamiltonian trajectories dynamically to maximize the efficiency of the exploration. That length, however, is capped at $2^{\text{max\_treedepth}}$ steps to prevent trajectories from growing without bound.

When numerical Hamiltonian trajectories are long but finite this truncation will limit the computational efficiency. Increasing `max_treedepth` allow the trajectories to expand further. While the resulting trajectories will be more expensive that added cost will be more than made up for by increased computational efficiency.

The energy fraction of missing information, or E-FMI, quantifies how well the Hamiltonian dynamics are able to explore the target distribution. If the E-FMI is too small then even the exact Hamiltonian trajectories will be limited to confined regions of the ambient space and full exploration will be possible only with the momenta resampling between trajectories. In this case the Markov chain exploration devolves into less efficient, diffusive behavior where Markov chain Monte Carlo estimation is fragile at best.

This confinement is caused by certain geometries in the target distribution, most commonly a funnel geometry where some subset of parameters shrink together as another parameter ranges across its typical values. The only way to avoid these problems is to identify the problematic geometry and then find a reparameterization of the ambient space that transforms the geometry into something more pleasant.

Finally the average proxy accept statistic is a summary for Stan's step size adaptation. During warmup the integrator step size is dynamically tuned until this statistic achieves the target value which defaults to 0.801. Because this adaptation is stochastic the realized average during the main sampling phase can often vary between 0.75 and 0.85.

So long as the target distribution is sufficiently well-behaved then the adaptation should always converge to that target, at least for long enough warmup periods. Small averages indicate some obstruction to the adaptation, for example discontinuities in the target distribution or inaccurate gradient evaluations.

```r
check_all_hmc_diagnostics <- function(fit,
                                      adapt_target=0.801,
                                      max_treedepth=10) {
  sampler_params <- get_sampler_params(fit, inc_warmup=FALSE)

  no_warning <- TRUE

  # Check divergences
  divergent <- do.call(rbind, sampler_params)[,'divergent__']
  n = sum(divergent)
  N = length(divergent)
```

```
if (n > 0) {
  no_warning <- FALSE
  cat(sprintf('%s of %s iterations ended with a divergence (%s%%).\n',
              n, N, 100 * n / N))

  cat('  Divergences are due unstable numerical integration.\n')
  cat('  These instabilities are often due to posterior degeneracies.\n')
  cat('  If there are only a small number of divergences then running\n')
  cat(sprintf('with adapt_delta larger than %.3f may reduce the divergences\n',
              adapt_target))
  cat('at the cost of more expensive transitions.\n\n')
}

# Check transitions that ended prematurely due to maximum tree depth limit
treedepths <- do.call(rbind, sampler_params)[,'treedepth__']
n = length(treedepths[sapply(treedepths, function(x) x >= max_treedepth)])
N = length(treedepths)

if (n > 0) {
  no_warning <- FALSE
  cat(sprintf('%s of %s iterations saturated the maximum tree depth of %s (%s%%).',
              n, N, max_treedepth, 100 * n / N))

  cat('  Increasing max_depth will increase the efficiency of the transitions.\n\n')
}

# Checks the energy fraction of missing information (E-FMI)
no_efmi_warning <- TRUE
for (c in 1:length(sampler_params)) {
  energies = sampler_params[c][[1]][,'energy__']
  numer = sum(diff(energies)**2) / length(energies)
  denom = var(energies)
  if (numer / denom < 0.2) {
    no_warning <- FALSE
    no_efmi_warning <- FALSE
    cat(sprintf('Chain %s: E-FMI = %s.\n', n, numer / denom))
  }
}
if (!no_efmi_warning) {
  cat('  E-FMI below 0.2 suggests a funnel-like geometry hiding\n')
  cat('somewhere in the posterior distribution.\n\n')
  }

  # Check convergence of the stepsize adaptation
  no_accept_warning <- TRUE
  for (c in 1:length(sampler_params)) {
    ave_accept_proxy <- mean(sampler_params[[c]][,'accept_stat__'])
    if (ave_accept_proxy < 0.9 * adapt_target) {
      no_warning <- FALSE
      no_accept_warning <- FALSE
      cat(sprintf('Chain %s: Average proxy acceptance statistic (%.3f)\n',
                  c, ave_accept_proxy))
      cat(sprintf('          is smaller than 90%% of the target (%.3f).\n',
                 adapt_target))
    }
  }
  if (!no_accept_warning) {
    cat('  A small average proxy acceptance statistic indicates that the\n')
    cat('integrator step size adaptation failed to converge.  This is often\n')
    cat('due to discontinuous or inexact gradients.\n\n')
  }

  if (no_warning) {
    cat('All Hamiltonian Monte Carlo diagnostic are consistent with\n')
    cat('accurate Markov chain Monte Carlo.\n\n')
  }
}
```

1.2 Integrator Inverse Metric Elements

Diagnostic failures indicate the presence of problems but only hint at the nature of those problems. In order to resolve the underlying problems we need to investigate them beyond these hints. Fortunately Hamiltonian Monte Carlo provides a wealth of additional information that can assist.

First we can look at the inverse metric adaptation in each of the Markov chains. Inconsistencies in the adapted inverse metric elements across the Markov chains are due to the individual chains encountering different behaviors during warmup.

```
plot_inv_metric <- function(fit, B=25) {
  chain_info <- get_adaptation_info(fit)
  C <- length(chain_info)
  inv_metric_elems <- list()
  for (c in 1:C) {
    raw_info <- chain_info[[c]]
    clean1 <- sub("# Adaptation terminated\n# Step size = [0-9.]*\n#",
                  "", raw_info)
    clean2 <- sub(" [a-zA-Z ]*:\n# ", "", clean1)
    clean3 <- sub("\n$", "", clean2)
    inv_metric_elems[[c]] <- as.numeric(strsplit(clean3, ',')[[1]])
  }

  min_elem <- min(unlist(inv_metric_elems))
  max_elem <- max(unlist(inv_metric_elems))

  delta <- (max_elem - min_elem) / B
  min_elem <- min_elem - delta
  max_elem <- max_elem + delta
  bins <- seq(min_elem, max_elem, delta)
  B <- B + 2

  max_y <- max(sapply(1:C, function(c)
    max(hist(inv_metric_elems[[c]], breaks=bins, plot=FALSE)$counts)))

  idx <- rep(1:B, each=2)
  x <- sapply(1:length(idx), function(b) if(b %% 2 == 1) bins[idx[b]]
                                           else bins[idx[b] + 1])

  par(mfrow=c(2, 2), mar = c(5, 2, 2, 1))
  colors <- c(c_dark, c_mid_highlight, c_mid, c_light_highlight)

  for (c in 1:C) {
    counts <- hist(inv_metric_elems[[c]], breaks=bins, plot=FALSE)$counts
    y <- counts[idx]

    plot(x, y, type="l", main=paste("Chain", c), col=colors[c],
         xlim=c(min_elem, max_elem), xlab="Inverse Metric Elements",
         ylim=c(0, 1.05 * max_y), ylab="", yaxt="n")
  }
}
```

1.3 Integrator Step Sizes

The other product of Stan's adaptation is the step size of the numerical integrator used to build the numerical Hamiltonian trajectories. As with the inverse metric elements heterogeneity in the adapted values across the Markov chains indicates that the Markov chains encountered substantially different behavior during warmup.

```r
display_stepsizes <- function(fit) {
  sampler_params <- get_sampler_params(fit, inc_warmup=FALSE)
  for (c in 1:4) {
    stepsize <- sampler_params[[c]][,'stepsize__'][1]
    cat(sprintf('Chain %s: Integrator Step Size = %f\n',
                c, stepsize))
  }
}
```

1.4 Numerical Trajectory Lengths

We can see the consequence of the adapted step sizes by looking at the numerical trajectories generated for each Hamiltonian Markov transition. The longer these trajectories the more degenerate the target distribution, and the more expensive it is to explore.

```r
plot_num_leapfrog <- function(fit) {
  sampler_params <- get_sampler_params(fit, inc_warmup=FALSE)

  max_n <- max(sapply(1:4, function(c) max(sampler_params[[c]][,'n_leapfrog__'])))
  max_count <- max(sapply(1:4, function(c) max(table(sampler_params[[c]][,'n_leapfrog__'])

  colors <- c(c_dark, c_mid_highlight, c_mid, c_light_highlight)

  idx <- rep(1:max_n, each=2)
  xs <- sapply(1:length(idx), function(b) if(b %% 2 == 0) idx[b] + 0.5
                                          else idx[b] - 0.5)

  par(mfrow=c(2, 2), mar = c(5, 5, 3, 1))

  for (c in 1:4) {
    stepsize <- round(sampler_params[[c]][,'stepsize__'][1], 3)

    counts <- hist(sampler_params[[c]][,'n_leapfrog__'],
                   seq(0.5, max_n + 0.5, 1), plot=FALSE)$counts
    pad_counts <- counts[idx]

    plot(xs, pad_counts, type="l",  lwd=2, col=colors[c],
         main=paste0("Chain ", c, " (Stepsize = ", stepsize, ")"),
         xlab="Numerical Trajectory Length", xlim=c(0.5, max_n + 0.5),
         ylab="", ylim=c(0, 1.1 * max_count), yaxt='n')
  }
}
```

1.5 Average Proxy Acceptance Statistic

When the different adaptation outcomes are due to problematic behaviors encountered during warmup then it the average proxy acceptance statistics should also vary across the Markov chains.

```
display_ave_accept_proxy <- function(fit) {
  sampler_params <- get_sampler_params(fit, inc_warmup=FALSE)

  for (c in 1:length(sampler_params)) {
    ave_accept_proxy <- mean(sampler_params[[c]][,'accept_stat__'])
    cat(sprintf('Chain %s: Average proxy acceptance statistic = %.3f\n',
                c, ave_accept_proxy))
  }
}
```

1.6 Divergence-Labeled Pairs Plot

One of the most powerful features of divergent transitions is that they not only indicate problematic geometry but also provide some spatial information on the source of that problematic geometry. In particular the states generated from unstable numerical Hamiltonian trajectories will tend to be closer to the problematic geometry than those from stable trajectories.

Consequently if we plot the states from divergent and non-divergent transitions separately then we should see the divergent states concentrate towards the problematic behavior. The high-dimensional states themselves can be visualized with pairs plots.

```
partition_div <- function(fit) {
  nom_params <- rstan:::extract(fit, permuted=FALSE)
  n_chains <- dim(nom_params)[2]
  params <- as.data.frame(do.call(rbind, lapply(1:n_chains, function(n) nom_params[,n,])))
  sampler_params <- get_sampler_params(fit, inc_warmup=FALSE)
  divergent <- do.call(rbind, sampler_params)[,'divergent__']
  params$divergent <- divergent

  div_params <- params[params$divergent == 1,]
  nondiv_params <- params[params$divergent == 0,]

  return(list(div_params, nondiv_params))
}
```

```
plot_div_pairs <- function(fit, names, transforms) {
  c_dark_trans <- c("#8F272780")
  c_green_trans <- c("#00FF0080")

  N <- length(names)
  N_plots <- choose(N, 2)

  if (N_plots <= 3) {
    par(mfrow=c(1, N_plots), mar = c(5, 5, 2, 1))
  } else if (N_plots == 4) {
    par(mfrow=c(2, 2), mar = c(5, 5, 2, 1))
  } else {
    par(mfrow=c(2, 3), mar = c(5, 5, 2, 1))
  }

  partition <- partition_div(fit)
  div_samples <- partition[[1]]
  nondiv_samples <- partition[[2]]

  for (n in 1:(N - 1)) {
    for (m in (n + 1):N) {

      name_x <- names[n]
      if (transforms[n] == 0) {
        x_nondiv_samples <- nondiv_samples[name_x][,1]
        x_div_samples <- div_samples[name_x][,1]
        x_name <- name_x
      } else if (transforms[n] == 1) {
        x_nondiv_samples <- log(nondiv_samples[name_x][,1])
        x_div_samples <- log(div_samples[name_x][,1])
```

```
          x_name <- paste("log(", name_x, ")", sep="")
        }
        xlims <- range(c(x_nondiv_samples, x_div_samples))

        name_y <- names[m]
        if (transforms[m] == 0) {
          y_nondiv_samples <- nondiv_samples[name_y][,1]
          y_div_samples <- div_samples[name_y][,1]
          y_name <- name_y
        } else if (transforms[m] == 1) {
          y_nondiv_samples <- log(nondiv_samples[name_y][,1])
          y_div_samples <- log(div_samples[name_y][,1])
          y_name <- paste("log(", name_y, ")", sep="")
        }
        ylims <- range(c(y_nondiv_samples, y_div_samples))

        plot(x_nondiv_samples, y_nondiv_samples,
             col=c_dark_trans, pch=16, main="",
             xlab=x_name, xlim=xlims, ylab=y_name, ylim=ylims)
        points(x_div_samples, y_div_samples,
               col=c_green_trans, pch=16)
      }
    }
  }
```

2 Expectand Diagnostic Functions

The Hamiltonian Monte Carlo diagnostics exploited the particular structure of the Hamiltonian Markov transition. For a general Markov transition we don't have any particular structure to exploit, and hence limited diagnostic options. In this general setting we have to investigate the behavior of not the entire state but instead particular expectands of interest.

2.1 khat

A Markov chain Monte Carlo central limit theorem cannot exist for the expectand $f : X \to \mathbb{R}$ unless both $\mathbb{E}_{\pi}[f]$ and $\mathbb{E}_{\pi}[f^2]$ are finite, in which case we say that the expectand is sufficiently integrable. Moreover the smaller the following moments the faster the central limit theorem kicks in.

$\hat{k}$ uses the tail behavior of a realized Markov chain to estimate the integrability of an expectand. More specifically $\hat{k}$ estimates the shape of a Pareto density function from non-central values of the expectand. If the tail behavior were exactly Pareto with shape parameter k then only the $(1/k)$-th order and lower moments would exist. For example with $k = 1$ the expectation $\mathbb{E}_{\pi}[f]$ is finite but $\mathbb{E}_{\pi}[f^2]$ is not, while for $k = \frac{1}{2}$ the expectations $\mathbb{E}_{\pi}[f]$ and $\mathbb{E}_{\pi}[f^2]$ are finite but $\mathbb{E}_{\pi}[f^3]$ is not.

The estimator $\hat{k}$ is constructed from the smallest and largest values of an expectand evaluated across a realized Markov chain, where the smallest and largest values are separated from the central values using a heuristic. Because $\hat{k}$ only estimates the tail shape I require a conservative threshold of $\hat{k} \geq 0.25$ for the diagnostic warning to be triggered.

If the expectand output is bounded then the lower and upper tail might consist of the same value. In this case the $\hat{k}$ estimator is poorly-behaved, but the boundedness also guarantees that moments of all orders exist. To make this diagnostic as robust as possible $\hat{k}$ will return -2 in these cases to avoid the diagnostic threshold.

```r
compute_khat <- function(fs) {
  N <- length(fs)
  sorted_fs <- sort(fs)

  if (sorted_fs[1] == sorted_fs[N]) {
    return (-2)
  }

  if (sorted_fs[1] < 0) {
    cat("x must be positive!")
    return (NA)
  }

  # Estimate 25% quantile
  q <- sorted_fs[floor(0.25 * N + 0.5)]

  if (q == sorted_fs[1]) {
    return (-2)
  }

  # Heurstic Pareto configuration
  M <- 20 + floor(sqrt(N))

  b_hat_vec <- rep(0, M)
  log_w_vec <- rep(0, M)
  for (m in 1:M) {
    b_hat_vec[m] <- 1 / sorted_fs[N] + (1 - sqrt(M / (m - 0.5))) / (3 * q)
    k_hat <- - mean( log(1 - b_hat_vec[m] * sorted_fs) )
    log_w_vec[m] <- N * ( log(b_hat_vec[m] / k_hat) + k_hat - 1)
  }

  max_log_w <- max(log_w_vec)
  b_hat <- sum(b_hat_vec * exp(log_w_vec - max_log_w)) /
    sum(exp(log_w_vec - max_log_w))

  mean( log (1 - b_hat * sorted_fs) )
}
```

```
compute_tail_khats <- function(fs) {
  f_center <- median(fs)
  fs_left <- abs(fs[fs < f_center] - f_center)
  fs_right <- fs[fs > f_center] - f_center

  # Default to 0 if left tail is ill-defined
  khat_left <- -2
  if (length(fs_left) > 0)
    khat_left <- compute_khat(fs_left)

  # Default to 0 if right tail is ill-defined
  khat_right <- -2
  if (length(fs_right) > 0)
    khat_right <- compute_khat(fs_right)

  c(khat_left, khat_right)
}
```

```
check_tail_khats <- function(unpermuted_samples) {
  no_warning <- TRUE
  for (c in 1:4) {
    fs <- unpermuted_samples[,c]
    khats <- compute_tail_khats(fs)
    if (khats[1] >= 0.25 & khats[2] >= 0.25) {
      cat(sprintf('Chain %s: Both left and right tail khats exceed 0.25!\n',
                  c))
      no_warning <- FALSE
    } else if (khats[1] < 0.25 & khats[2] >= 0.25) {
      cat(sprintf('Chain %s: Right tail khat exceeds 0.25!\n', c))
      no_warning <- FALSE
    } else if (khats[1] >= 0.25 & khats[2] < 0.25) {
      cat(sprintf('Chain %s: Left tail khat exceeds 0.25!\n', c))
      no_warning <- FALSE
    }
  }
  if (no_warning) {
    cat('Expectand appears to be sufficiently integrable.\n\n')
  } else {
    cat('  Large tail khats suggest that the expectand might\n')
    cat('not be sufficiently integrable.\n\n')
  }
}
```

2.2 Frozen Chains

Another sign of problems is when all evaluations of an expectand are constant. This could be due to the Markov chain being stuck at a single state or just that the pushforward distribution of the expectand concentrates on a single value. We can't distinguish between these possibilities without more information, but we can signal a constant expectand by looking at its empirical variance.

Here we'll use a Welford accumulator to compute the empirical variance of the expectand values in a single sweep.

```
welford_summary <- function(fs) {
  mean <- 0
  var <- 0

  N <- length(fs)
  for (n in 1:N) {
    delta <- fs[n] - mean
    mean <- mean + delta / n
    var <- var + delta * (fs[n] - mean)
  }

  var <- var/ (N - 1)

  return(c(mean, var))
}
```

```r
check_variances <- function(unpermuted_samples) {
  no_warning <- TRUE
  for (c in 1:4) {
    fs <- unpermuted_samples[,c]
    var <- welford_summary(fs)[2]
    if (var < 1e-10) {
      cat(sprintf('Chain %s: Expectand is constant!\n', c))
      no_warning <- FALSE
    }
  }
  if (no_warning) {
    cat('Expectand is varying in all Markov chains.')
  } else {
    cat('  If the expectand is not expected (haha) to be\n')
    cat('constant then the Markov transitions are misbehaving.\n')
  }
}
```

2.3 Split Rhat

One of the key features of Markov chain equilibrium is that the distribution of Markov chain realizations is independent of the initialization. In particular the expectand evaluations from any equilibrated Markov chain should be statistically equivalent to any other. Even more the evaluations across any subset of Markov chain states should be equivalent.

The split $\hat{R}$ statistic quantifies the heterogeneity in the expectand evaluations across an ensemble of Markov chains, each of which has been split in half. Mathematically split $\hat{R}$ is similar to analysis of variance in that compares the empirical variance of the average expectand values in each chain half to the average of the empirical variances in each chain half; the key difference is that split $\hat{R}$ transforms this ratio so that in equilibrium the statistic decays towards 1 from above.

When split $\hat{R}$ is much larger than 1 the expectand evaluations across each Markov chain halves are not consistent with each other. This could be because the Markov chains have not converged to the same typical set or because they have not yet expanded into that typical set.

```r
split_chain <- function(chain) {
  N <- length(chain)
  M <- N %/% 2
  list(chain1 <- chain[1:M], chain2 <- chain[(M + 1):N])
}
```

```
compute_split_rhat <- function(chains) {
  split_chains <- unlist(lapply(chains, function(c) split_chain(c)),
                         recursive=FALSE)

  N_chains <- length(split_chains)
  N <- sum(sapply(chains, function(c) length(c)))

  means <- rep(0, N_chains)
  vars <- rep(0, N_chains)

  for (c in 1:N_chains) {
    summary <- welford_summary(split_chains[[c]])
    means[c] <- summary[1]
    vars[c] <- summary[2]
  }

  total_mean <- sum(means) / N_chains
  W = sum(vars) / N_chains
  B = N * sum(sapply(means, function(m)
                            (m - total_mean)**2)) / (N_chains - 1)

  rhat = NaN
  if (abs(W) > 1e-10)
    rhat = sqrt( (N - 1 + B / W) / N )

  (rhat)
}
```

```
compute_split_rhats <- function(fit, expectand_idxs=NULL) {
  unpermuted_samples <- rstan:::extract(fit, permute=FALSE)

  input_dims <- dim(unpermuted_samples)
  N <- input_dims[1]
  C <- input_dims[2]
  I <- input_dims[3]

  if (is.null(expectand_idxs)) {
    expectand_idxs <- 1:I
  }
  bad_idxs <- setdiff(expectand_idxs, 1:I)
  if (length(bad_idxs) > 0) {
    cat(sprintf('Excluding the invalid expectand indices: %s\n',
                bad_idxs))
    expectand_idxs <- setdiff(expectand_idxs, bad_idxs)
  }

  rhats <- c()
  for (idx in expectand_idxs) {
    chains <- lapply(1:C, function(c) unpermuted_samples[,c,idx])
    rhats <- c(rhats, compute_split_rhat(chains))
  }
  return(rhats)
}
```

```
check_rhat <- function(unpermuted_samples) {
  chains <- lapply(1:4, function(c) unpermuted_samples[,c])
  rhat <- compute_split_rhat(chains)

  no_warning <- TRUE
  if (is.nan(rhat)) {
    cat('All Markov chains are frozen!\n')
  } else if (rhat > 1.1) {
    cat(sprintf('Split rhat is %f!\n', rhat))
    no_warning <- FALSE
  }
  if (no_warning) {
    cat('Markov chains are consistent with equilibrium.\n')
  } else {
    cat('  Split rhat larger than 1.1 is inconsistent with equilibrium.\n')
  }
}
```

2.4 Integrated Autocorrelation Time

The information about the target distribution encoded within a Markov chain, and hence the potential precision of Markov chain Monte Carlo estimators, is limited by the autocorrelation of the internal states. Assuming equilibrium we can estimate the stationary autocorrelations between the outputs of a given expectand from the realized Markov chain and then combine them into an estimate of the integrated autocorrelation time which moderates the asymptotic variance of well-behaved Markov chain Monte Carlo estimators.

If this empirical integrated autocorrelation time is a substantial proportion of the length of the realized Markov chain then there won't be enough information to supply robust Markov chain Monte Carlo estimators. Here I set the diagnostic warning to a quarter of the total number of iterations.

When Markov chains have not equilibrated the empirical autocorrelation time will not longer be related to the error of Markov chain Monte Carlo estimators. That said it still provides a useful quantification of the autocorrelations within a realized Markov chain. In particular it provides a useful way to distinguish if some diagnostic failures are due to Markov chains that are just too short or more persistent problems.

```
compute_int_ac_time <- function(fs) {
  # Compute empirical autocorrelations
  N <- length(fs)
  zs <- fs - mean(fs)

  B <- 2**ceiling(log2(N)) # Next power of 2 after N
  zs_buff <- c(zs, rep(0, B - N))

  Fs <- fft(zs_buff)
  Ss <- Fs * Conj(Fs)
  Rs <- fft(Ss, inverse=TRUE)

  acov_buff <- Re(Rs)
  rhos <- head(acov_buff, N) / acov_buff[1]

  # Drop last lag if (L + 1) is odd so that the lag pairs are complete
  L <- N
  if ((L + 1) %% 2 == 1)
    L <- L - 1

  # Number of lag pairs
  P <- (L + 1) / 2

  # Construct asymptotic correlation from initial monotone sequence
  old_pair_sum <- rhos[1] + rhos[2]
  for (p in 2:P) {
    current_pair_sum <- rhos[2 * p - 1] + rhos[2 * p]

    if (current_pair_sum < 0) {
      rho_sum <- sum(rhos[2:(2 * p)])

      if (rho_sum <= -0.25)
        rho_sum <- -0.25

      asymp_corr <- 1.0 + 2 * rho_sum
      return (asymp_corr)
    }

    if (current_pair_sum > old_pair_sum) {
      current_pair_sum <- old_pair_sum
      rhos[2 * p - 1] <- 0.5 * old_pair_sum
      rhos[2 * p] <- 0.5 * old_pair_sum
    }

    if (p == P) {
      # throw some kind of error when autocorrelation
      # sequence doesn't get terminated
    }

    old_pair_sum <- current_pair_sum
  }
}
```

```
compute_min_int_ac_times <- function(fit, expectand_idxs=NULL) {
  unpermuted_samples <- rstan:::extract(fit, permute=FALSE)

  input_dims <- dim(unpermuted_samples)
  N <- input_dims[1]
  C <- input_dims[2]
  I <- input_dims[3]

  expectand_names <- names(unpermuted_samples[1,1,])

  if (is.null(expectand_idxs)) {
    expectand_idxs <- 1:I
  }

  bad_idxs <- setdiff(expectand_idxs, 1:I)
  if (length(bad_idxs) > 0) {
    cat(sprintf('Excluding the invalid expectand indices: %s',
                bad_idxs))
    expectand_idxs <- setdiff(expectand_idxs, bad_idxs)
  }

  min_int_ac_times <- c()

  for (idx in expectand_idxs) {
    int_ac_times <- rep(0, C)
    for (c in 1:C) {
      fs <- unpermuted_samples[,c, idx]
      int_ac_times[c] <- compute_int_ac_time(fs)
    }
    min_int_ac_times <- c(min_int_ac_times, min(int_ac_times))
  }
  return(min_int_ac_times)
}
```

```
  check_int_ac_time <- function(unpermuted_samples) {
    N <- dim(unpermuted_samples)[1]
    no_warning <- TRUE
    for (c in 1:4) {
      fs <- unpermuted_samples[,c]
      int_ac_time <- compute_int_ac_time(fs)
      if (int_ac_time / N > 0.25) {
        cat(sprintf('Chain %s: The integrated autocorrelation time', c))
        cat('exceeds 0.25 * N!\n')
        no_warning <- FALSE
      }
    }
    if (no_warning) {
      cat('Autocorrelations within each Markov chain appear to be reasonable.\n')
    } else {
      cat('  Autocorrelations in at least one Markov chain are large enough')
      cat('that Markov chain Monte Carlo estimates may not be reliable.\n')
    }
  }
```

Assuming stationarity we can use the empirical integrated autocorrelation time to estimate the effective sample size, and hence the Markov chain Monte Carlo standard error, for any well-behaved expectand estimator

$$\hat{f} \approx \mathbb{E}_{\pi}[f].$$

The desired effective sample size depends on the precision required for a given Markov chain Monte Carlo estimator. This can vary not only from analysis to analysis but also between multiple expectands within a single analysis. That said an effective sample size of 100 is sufficient for most applications and provides a useful rule of thumb.

As with the empirical integrated autocorrelation times we have to be careful with the empirical effective sample sizes. We can construct these estimators from any Markov chain, but if that chain hasn't reach equilibrium then these estimators will have no connection to Markov chain Monte Carlo error quantification!

```
check_neff <- function(unpermuted_samples,
                       min_neff_per_chain=100) {
  N <- dim(unpermuted_samples)[1]
  no_warning <- TRUE
  for (c in 1:4) {
    fs <- unpermuted_samples[,c]
    int_ac_time <- compute_int_ac_time(fs)
    neff <- N / int_ac_time
    if (neff < min_neff_per_chain) {
      cat(sprintf('Chain %s: The effective sample size %f is too small!\n',
                  c, neff))
      no_warning <- FALSE
    }
  }
  if (no_warning) {
    cat('All effective sample sizes are sufficiently large.\n')
  } else {
    cat('  If the effective sample size is too small then\n')
    cat('Markov chain Monte Carlo estimators will be imprecise.\n\n')
  }
}
```

2.5 All Expectand Diagnostics

In practice we have no reason not to check all of these diagnostics at once for each expectand of interest.

```
check_all_expectand_diagnostics <- function(fit,
                                            expectand_idxs=NULL,
                                            min_neff_per_chain=100) {
  unpermuted_samples <- rstan:::extract(fit, permute=FALSE)
```

```r
input_dims <- dim(unpermuted_samples)
N <- input_dims[1]
C <- input_dims[2]
I <- input_dims[3]

expectand_names <- names(unpermuted_samples[1,1,])

if (is.null(expectand_idxs)) {
  expectand_idxs <- 1:I
}

bad_idxs <- setdiff(expectand_idxs, 1:I)
if (length(bad_idxs) > 0) {
  cat(sprintf('Excluding the invalid expectand indices: %s',
              bad_idxs))
  expectand_idxs <- setdiff(expectand_idxs, bad_idxs)
}

no_khat_warning <- TRUE
no_zvar_warning <- TRUE
no_rhat_warning <- TRUE
no_tauhat_warning <- TRUE
no_neff_warning <- TRUE

message <- ""

for (idx in expectand_idxs) {

  local_warning <- FALSE
  local_message <- paste0(expectand_names[idx], ':\n')

  for (c in 1:C) {
    # Check tail khats in each Markov chain
    fs <- unpermuted_samples[,c, idx]
    khats <- compute_tail_khats(fs)
    if (khats[1] >= 0.25 & khats[2] >= 0.25) {
      no_khat_warning <- FALSE
      local_warning <- TRUE
      local_message <-
        paste0(local_message,
               sprintf('  Chain %s: Both left and right tail hat{k}s', c),
              sprintf('(%.3f, %.3f) exceed 0.25!\n', khats[1], khats[2]))
    } else if (khats[1] < 0.25 & khats[2] >= 0.25) {
      no_khat_warning <- FALSE
      local_warning <- TRUE
      local_message <-
        paste0(local_message,
               sprintf('  Chain %s: Right tail hat{k} (%.3f) exceeds 0.25!\n',
                       c, khats[2]))
    } else if (khats[1] >= 0.25 & khats[2] < 0.25) {
      no_khat_warning <- FALSE
      local_warning <- TRUE
      local_message <-
        paste0(local_message,
               sprintf('  Chain %s: Left tail hat{k} ($.3f) exceeds 0.25!\n',
               c, khats[1]))
    }

    # Check empirical variance in each Markov chain
    var <- welford_summary(fs)[2]
    if (var < 1e-10) {
      no_zvar_warning <- FALSE
      local_warning <- TRUE
      local_message <-
        paste0(local_message,
               sprintf('Chain %s: Expectand has vanishing', c),
               ' empirical variance!\n')
    }
  }

  # Check split Rhat across Markov chains
  chains <- lapply(1:C, function(c) unpermuted_samples[,c,idx])
  rhat <- compute_split_rhat(chains)

  if (is.nan(rhat)) {
    local_message <- paste0(local_message,
                            '  Split hat{R} is ill-defined!\n')
  } else if (rhat > 1.1) {
    no_rhat_warning <- FALSE
    local_warning <- TRUE
    local_message <-
      paste0(local_message,
              sprintf('  Split hat{R} (%.3f) exceeds 1.1!\n', rhat))
  }

  for (c in 1:C) {
    # Check empirical integrated autocorrelation time
    fs <- unpermuted_samples[,c, idx]
    int_ac_time <- compute_int_ac_time(fs)
    if (int_ac_time / N > 0.25) {
      no_tauhat_warning <- FALSE
      local_warning <- TRUE
      local_message <-
        paste0(local_message,
               sprintf('  Chain %s: hat{tau} per iteration (%.3f)',
                       c, int_ac_time / N),
               ' exceeds 0.25!\n')
    }

    # Check empirical effective sample size
    neff <- N / int_ac_time
    if (neff < min_neff_per_chain) {
      no_neff_warning <- FALSE
      local_warning <- TRUE
      local_message <-
        paste0(local_message,
               sprintf('  Chain %s: hat{ESS} (%.3f) is smaller than',
                       c, neff),
               sprintf(' desired (%s)!\n', min_neff_per_chain))
    }
  }
  local_message <- paste0(local_message, '\n')
  if (local_warning) {
    message <- paste0(message, local_message)
  }
}

if (!no_khat_warning) {
  message <- paste0(message,
                    'Large tail hat{k}s suggest that the expectand',
                    ' might not be sufficiently integrable.\n\n')
}
if (!no_zvar_warning) {
    message <- paste0(message,
                      'Zero empirical variance suggests that the Markov',
                      ' transitions are misbehaving.\n\n')
  }
  if (!no_rhat_warning) {
    message <- paste0(message,
                      'Split hat{R} larger than 1.1 is inconsisent with', ' equilibrium.\n
  }
  if (!no_tauhat_warning) {
    message <- paste0(message,
                      'hat{tau} larger than a quarter of the Markov chain',
                      ' length suggests that Markov chain Monte Carlo,',
                      ' estimates will be unreliable.\n\n')
  }
  if (!no_neff_warning) {
    message <- paste0(message,
                      'If hat{ESS} is too small then Markov chain',
                      ' Monte Carlo estimators will be too imprecise.\n\n')
  }

  if(no_khat_warning & no_zvar_warning & no_rhat_warning & no_tauhat_warning & no_neff_war
    message <- paste0('All expectands checked appear to be behaving',
                      'well enough for Markov chain Monte Carlo estimation.\n')
  }

  cat(message)
}
```

```
expectand_diagnostics_summary <- function(fit,
                                          expectand_idxs=NULL,
                                          min_neff_per_chain=100) {
  unpermuted_samples <- rstan:::extract(fit, permute=FALSE)

  input_dims <- dim(unpermuted_samples)
  N <- input_dims[1]
  C <- input_dims[2]
  I <- input_dims[3]

  if (is.null(expectand_idxs)) {
    expectand_idxs <- 1:I
  }
```

```
bad_idxs <- setdiff(expectand_idxs, 1:I)
if (length(bad_idxs) > 0) {
  cat(sprintf('Excluding the invalid expectand indices: %s',
              bad_idxs))
  expectand_idxs <- setdiff(expectand_idxs, bad_idxs)
}

failed_idx <- c()
failed_khat_idx <- c()
failed_zvar_idx <- c()
failed_rhat_idx <- c()
failed_tauhat_idx <- c()
failed_neff_idx <- c()

for (idx in expectand_idxs) {

  for (c in 1:C) {
    # Check tail khats in each Markov chain
    fs <- unpermuted_samples[,c, idx]
    khats <- compute_tail_khats(fs)
    if (khats[1] >= 0.25 | khats[2] >= 0.25) {
      failed_idx <- c(failed_idx, idx)
      failed_khat_idx <- c(failed_khat_idx, idx)
    }

    # Check empirical variance in each Markov chain
    var <- welford_summary(fs)[2]
    if (var < 1e-10) {
      failed_idx <- c(failed_idx, idx)
      failed_zvar_idx <- c(failed_zvar_idx, idx)
    }
  }

  # Check split Rhat across Markov chains
  chains <- lapply(1:C, function(c) unpermuted_samples[,c,idx])
  rhat <- compute_split_rhat(chains)

  if (rhat > 1.1) {
    failed_idx <- c(failed_idx, idx)
    failed_rhat_idx <- c(failed_rhat_idx, idx)
  }
```

```
    for (c in 1:C) {
      # Check empirical integrated autocorrelation time
      fs <- unpermuted_samples[,c, idx]
      int_ac_time <- compute_int_ac_time(fs)
      if (int_ac_time / N > 0.25) {
        failed_idx <- c(failed_idx, idx)
        failed_tauhat_idx <- c(failed_tauhat_idx, idx)
      }

      # Check empirical effective sample size
      neff <- N / int_ac_time
      if (neff < min_neff_per_chain) {
        failed_idx <- c(failed_idx, idx)
        failed_neff_idx <- c(failed_neff_idx, idx)
      }
    }
  }

  failed_idx <- unique(failed_idx)
  if (length(failed_idx)) {
    cat(sprintf('The expectands %s triggered diagnostic warnings.\n\n',
                paste(failed_idx, collapse=", ")))
  } else {
    cat(paste0('All expectands checked appear to be behaving',
               'well enough for Markov chain Monte Carlo estimation.\n'))
  }

  failed_khat_idx <- unique(failed_khat_idx)
  if (length(failed_khat_idx)) {
    cat(sprintf('The expectands %s triggered hat{k} warnings.\n',
                paste(failed_khat_idx, collapse=", ")))
    cat(paste0('  Large tail hat{k}s suggest that the expectand',
               ' might not be sufficiently integrable.\n\n'))
  }

  failed_zvar_idx <- unique(failed_zvar_idx)
  if (length(failed_zvar_idx)) {
    cat(sprintf('The expectands %s triggered zero variance warnings.\n',
                paste(failed_zvar_idx, collapse=", ")))
    cat(paste0('  Zero empirical variance suggests that the Markov',
               ' transitions are misbehaving.\n\n'))
  }

  failed_rhat_idx <- unique(failed_rhat_idx)
  if (length(failed_rhat_idx)) {
    cat(sprintf('The expectands %s triggered hat{R} warnings.\n',
                paste(failed_rhat_idx, collapse=", ")))
    cat(paste0('  Split hat{R} larger than 1.1 is inconsistent with',
               ' equilibrium.\n\n'))
  }

  failed_tauhat_idx <- unique(failed_tauhat_idx)
  if (length(failed_tauhat_idx)) {
    cat(sprintf('The expectands %s triggered hat{tau} warnings.\n',
                paste(failed_tauhat_idx, collapse=", ")))
    cat(paste0('  hat{tau} larger than a quarter of the Markov chain',
               ' length suggests that Markov chain Monte Carlo,',
               ' estimates will be unreliable.\n\n'))
  }

  failed_neff_idx <- unique(failed_neff_idx)
  if (length(failed_neff_idx)) {
    cat(sprintf('The expectands %s triggered hat{ESS} warnings.\n',
                paste(failed_neff_idx, collapse=", ")))
    cat(paste0('  If hat{ESS} is too small then Markov chain',
               ' Monte Carlo estimators will be too imprecise.\n\n'))
  }
}
```

2.6 Empirical Autocorrelation Visualization

If we encounter large empirical integrated autocorrelation times, or small estimated effective sample sizes, then we may want to follow up with the empirical autocorrelations themselves. An empirical correlogram provides a useful visualization of these estimates.

```
compute_rhos <- function(fs) {
  # Need to check for zero variance first?

  # Compute empirical autocorrelations
  N <- length(fs)
  zs <- fs - mean(fs)
```

```r
B <- 2**ceiling(log2(N)) # Next power of 2 after N
zs_buff <- c(zs, rep(0, B - N))

Fs <- fft(zs_buff)
Ss <- Fs * Conj(Fs)
Rs <- fft(Ss, inverse=TRUE)

acov_buff <- Re(Rs)
rhos <- head(acov_buff, N) / acov_buff[1]

# Drop last lag if (L + 1) is odd so that the
# lag pairs are complete
L <- N
if ((L + 1) %% 2 == 1)
  L <- L - 1

# Number of lag pairs
P <- (L + 1) / 2

# Construct asymptotic correlation from initial monotone sequence
old_pair_sum <- rhos[1] + rhos[2]
max_L <- N

for (p in 2:P) {
  current_pair_sum <- rhos[2 * p - 1] + rhos[2 * p]

  if (current_pair_sum < 0) {
    max_L <- 2 * p
    rhos[(max_L + 1):N] <- 0
    break
  }

  if (current_pair_sum > old_pair_sum) {
    current_pair_sum <- old_pair_sum
    rhos[2 * p - 1] <- 0.5 * old_pair_sum
    rhos[2 * p] <- 0.5 * old_pair_sum
  }

  old_pair_sum <- current_pair_sum
}
return(rhos)
```

```
}
```

```
plot_empirical_correlogram <- function(unpermuted_fs,
                                       max_L,
                                       rholim=c(-0.2, 1.1),
                                       name="") {
  idx <- rep(0:max_L, each=2)
  xs <- sapply(1:length(idx), function(b) if(b %% 2 == 0) idx[b] + 0.5
                                          else idx[b] - 0.5)

  plot(0, type="n", main=name,
       xlab="Lag", xlim=c(-0.5, max_L + 0.5),
       ylab="Empirical Autocorrelation", ylim=rholim)
  abline(h=0, col="#DDDDDD", lty=2, lwd=2)

  colors <- c(c_dark, c_mid_highlight, c_mid, c_light_highlight)
  for (c in 1:4) {
    fs <- unpermuted_fs[,c]
    rhos <- compute_rhos(fs)
    pad_rhos <- unlist(lapply(idx, function(n) rhos[n + 1]))
    lines(xs, pad_rhos, lwd=2, col=colors[c])
  }
}
```

2.7 Chain-Separated Pairs Plot

We can also visualize strong autocorrelations by coloring the states of each Markov chain in a continuous gradient. When neighboring states are strongly correlated these colors will appear to vary smoothly across the ambient space. More productive Markov transitions result in a more chaotic spray of colors.

```
library(colormap)

plot_chain_sep_pairs <- function(unpermuted_f1s, name_x,
                                 unpermuted_f2s, name_y) {
  N <- dim(unpermuted_f1s)[1]

  nom_colors <- c("#DCBCBC", "#C79999", "#B97C7C",
                  "#A25050", "#8F2727", "#7C0000")
  cmap <- colormap(colormap=nom_colors, nshades=N)
  min_x <- min(sapply(1:4, function(c) min(unpermuted_f1s[,c])))
  max_x <- max(sapply(1:4, function(c) max(unpermuted_f1s[,c])))

  min_y <- min(sapply(1:4, function(c) min(unpermuted_f2s[,c])))
  max_y <- max(sapply(1:4, function(c) max(unpermuted_f2s[,c])))

  par(mfrow=c(2, 2), mar = c(5, 5, 3, 1))

  for (c in 1:4) {
    plot(0, type="n", main=paste("Chain", c),
         xlab=name_x, xlim=c(min_x, max_x),
         ylab=name_y, ylim=c(min_y, max_y))

    points(unlist(lapply(1:4, function(c) unpermuted_f1s[,c])),
           unlist(lapply(1:4, function(c) unpermuted_f2s[,c])),
           col="#DDDDDD", pch=16, cex=1.0)
    points(unpermuted_f1s[,c], unpermuted_f2s[,c],
         col=cmap, pch=16, cex=1.0)
  }
}
```

3 Markov chain Monte Carlo Estimation

If none of the diagnostics indicate an obstruction to a Markov chain Monte Carlo central limit theorem then we can construct expectation value estimates and their standard errors.

```
pushforward_chains <- function(chains, expectand) {
  lapply(chains, function(c) sapply(c, function(x) expectand(x)))
}
```

```
mcmc_est <- function(fs) {
  N <- length(fs)
  if (N == 1) {
    return(c(fs[1], 0, NaN))
  }

  summary <- welford_summary(fs)

  if (summary[2] == 0) {
    return(c(summary[1], 0, NaN))
  }

  int_ac_time <- compute_int_ac_time(fs)
  neff <- N / int_ac_time
  return(c(summary[1], sqrt(summary[2] / neff), neff))
}
```

```
ensemble_mcmc_est <- function(chains) {
  C <- length(chains)
  chain_ests <- lapply(chains, function(c) mcmc_est(c))

  # Total effective sample size
  total_ess <- sum(sapply(chain_ests, function(est) est[3]))

  if (is.nan(total_ess)) {
    m <- mean(sapply(chain_ests, function(est) est[1]))
    se <- mean(sapply(chain_ests, function(est) est[2]))
    return (c(m, se, NaN))
  }

  # Ensemble average weighted by effective sample size
  mean <- sum(sapply(chain_ests,
                     function(est) est[3] * est[1])) / total_ess

  # Ensemble variance weighed by effective sample size
  # including correction for the fact that individual Markov chain
  # variances are defined relative to the individual mean estimators
  # and not the ensemble mean estimator
  vars <- rep(0, C)

  for (c in 1:C) {
    est <- chain_ests[[c]]
    chain_var <- est[3] * est[2]**2
    var_update <- (est[1] - mean)**2
    vars[c] <- est[3] * (var_update + chain_var)
  }
  var <- sum(vars) / total_ess

  c(mean, sqrt(var / total_ess), total_ess)
}
```

In addition to examining the single expectation value of an expectand we can also visualize the entire pushforward distribution of the expectand by estimating the target probabilities in histogram bins.

```
plot_pushforward_hist <- function(unpermuted_samples, B, name="f") {
  mean_p <- rep(0, B)
  delta_p <- rep(0, B)

  min_f <- min(unpermuted_samples)
  max_f <- max(unpermuted_samples)

  # Add bounding bins
  delta <- (max_f - min_f) / B
  min_f <- min_f - delta
  max_f <- max_f + delta
  bins <- seq(min_f, max_f, delta)
  B <- B + 2

  chains <- lapply(1:4, function(c) unpermuted_samples[,c])

  for (b in 1:B) {
    bin_indicator <- function(x) {
      ifelse(bins[b] <= x & x < bins[b + 1], 1, 0)
    }
    indicator_chains <- pushforward_chains(chains, bin_indicator)
    est <- ensemble_mcmc_est(indicator_chains)

    # Normalize bin probabilities by bin width to allow
    # for direct comparison to probability density functions
    width = bins[b + 1] - bins[b]
    mean_p[b] = est[1] / width
    delta_p[b] = est[2] / width
  }

  idx <- rep(1:B, each=2)
  x <- sapply(1:length(idx), function(b) if(b %% 2 == 1) bins[idx[b]]
                                         else bins[idx[b] + 1])
  lower_inter <- sapply(idx, function (n)
                              max(mean_p[n] - 2 * delta_p[n], 0))
  upper_inter <- sapply(idx, function (n)
                              min(mean_p[n] + 2 * delta_p[n], 1 / width))
  min_y <- min(lower_inter)
  max_y <- max(1.05 * upper_inter)

  plot(1, type="n", main="",
       xlim=c(min_f, max_f), xlab=name,
       ylim=c(min_y, max_y), ylab="", yaxt="n")
  title(ylab="Estimated Bin\nProbabilities", mgp=c(1, 1, 0))

  polygon(c(x, rev(x)), c(lower_inter, rev(upper_inter)),
          col = c_light, border = NA)
  lines(x, mean_p[idx], col=c_dark, lwd=2)
}
```

4 Demonstration

Now let's put all of these analysis tools to use with an `rstan` fit object.

First we setup our local `R` environment.

```
library(rstan)
rstan_options(auto_write = TRUE)            # Cache compiled Stan programs
options(mc.cores = parallel::detectCores()) # Parallelize chains
parallel:::setDefaultClusterOptions(setup_strategy = "sequential")
```

Then we can simulate some binary data from a logistic regression model.

simu_logistic_reg.stan

```
transformed data {
  int<lower=0> M = 3;          // Number of covariates
  int<lower=0> N = 1000;       // Number of observations

  vector[M] x0 = [-1, 0, 1]'; // Covariate baseline
  vector[M] z0 = [-3, 1, 2]'; // Latent functional behavior baseline
  real gamma0 = -2.6;                     // True intercept
  vector[M] gamma1 = [0.2, -2.0, 0.33]';   // True slopes
  matrix[M, M] gamma2 = [ [+0.40, -0.05, -0.20],
                          [-0.05, -1.00, -0.05],
                          [-0.20, -0.05, +0.50] ];
}
```

```
generated quantities {
  matrix[N, M] X; // Covariate design matrix
  real y[N];      // Variates

  for (n in 1:N) {
    real x2 = -5;
    while (x2 < x0[2] - 4 || x2 > x0[2] + 4)
      x2 = normal_rng(x0[2], 2);

    X[n, 2] = x2;
    X[n, 1] = normal_rng(x0[1] + 1.0 * cos(1.5 * (X[n, 2] - x0[2])), 0.3);
    X[n, 3] = normal_rng(x0[3] + 0.76 * (X[n, 1] - x0[1]), 0.5);

    y[n] = bernoulli_logit_rng(  gamma0
                               + (X[n] - z0') * gamma1
                               + (X[n] - z0') * gamma2 * (X[n] - z0')');
  }
}
```

```
simu <- stan(file="stan_programs/simu_logistic_reg.stan",
             iter=1, warmup=0, chains=1,
             seed=4838282, algorithm="Fixed_param")
```

```
SAMPLING FOR MODEL 'simu_logistic_reg' NOW (CHAIN 1).
Chain 1: Iteration: 1 / 1 [100%]  (Sampling)
Chain 1:
Chain 1:  Elapsed Time: 0 seconds (Warm-up)
Chain 1:                0.00175 seconds (Sampling)
Chain 1:                0.00175 seconds (Total)
Chain 1:
```

```
X <- extract(simu)$X[1,,]
y <- extract(simu)$y[1,]

data <- list("M" = 3, "N" = 1000, "x0" = c(-1, 0, 1), "X" = X, "y" = y)
```

We'll try to fit this model not with a constraint-respecting logistic regression model but rather a constraint blaspheming linear probability model. Importantly the resulting posterior density function is discontinuous with configurations `alpha + deltaX * beta > 0` resulting in finite bernoulli_lpmf outputs and those with alpha + deltaX * beta <= 0 resulting in minus infinite outputs.

bernoulli_linear.stan

```
data {
  int<lower=0> M; // Number of covariates
  int<lower=0> N; // Number of observations

  vector[M] x0;   // Covariate baselines
  matrix[N, M] X; // Covariate design matrix

  int<lower=0, upper=1> y[N]; // Variates
}

transformed data {
  matrix[N, M] deltaX;
  for (n in 1:N) {
    deltaX[n,] = X[n] - x0';
  }
}

parameters {
  real alpha;       // Intercept
  vector[M] beta;   // Linear slopes
}

model {
  // Prior model
  alpha ~ normal(0, 1);
  beta ~ normal(0, 1);

  // Vectorized observation model
  y ~ bernoulli(alpha + deltaX * beta);
}

// Simulate a full observation from the current value of the parameters
generated quantities {
  vector[N] p = alpha + deltaX * beta;
  int y_pred[N] = bernoulli_rng(p);
}
```

Because of this awkward constraint we have to carefully initialize our Markov chains to satisfy the `alpha + deltaX * beta > 0` constraint.

```
set.seed(48383499)

interval_inits <- list()

for (c in 1:4) {
  beta <- c(0, 0, 0)
  alpha <- rnorm(1, 0.5, 0.1)
  interval_inits[[c]] <- list("alpha" = alpha, "beta" = beta)
}

fit <- stan(file="stan_programs/bernoulli_linear.stan",
            data=data, seed=8438338,
            warmup=1000, iter=2024, refresh=0,
            init=interval_inits)
```

Stan is able to run to completion, but just how useful are the Markov chains that it generates?

Let's start with the Hamiltonian Monte Carlo diagnostics.

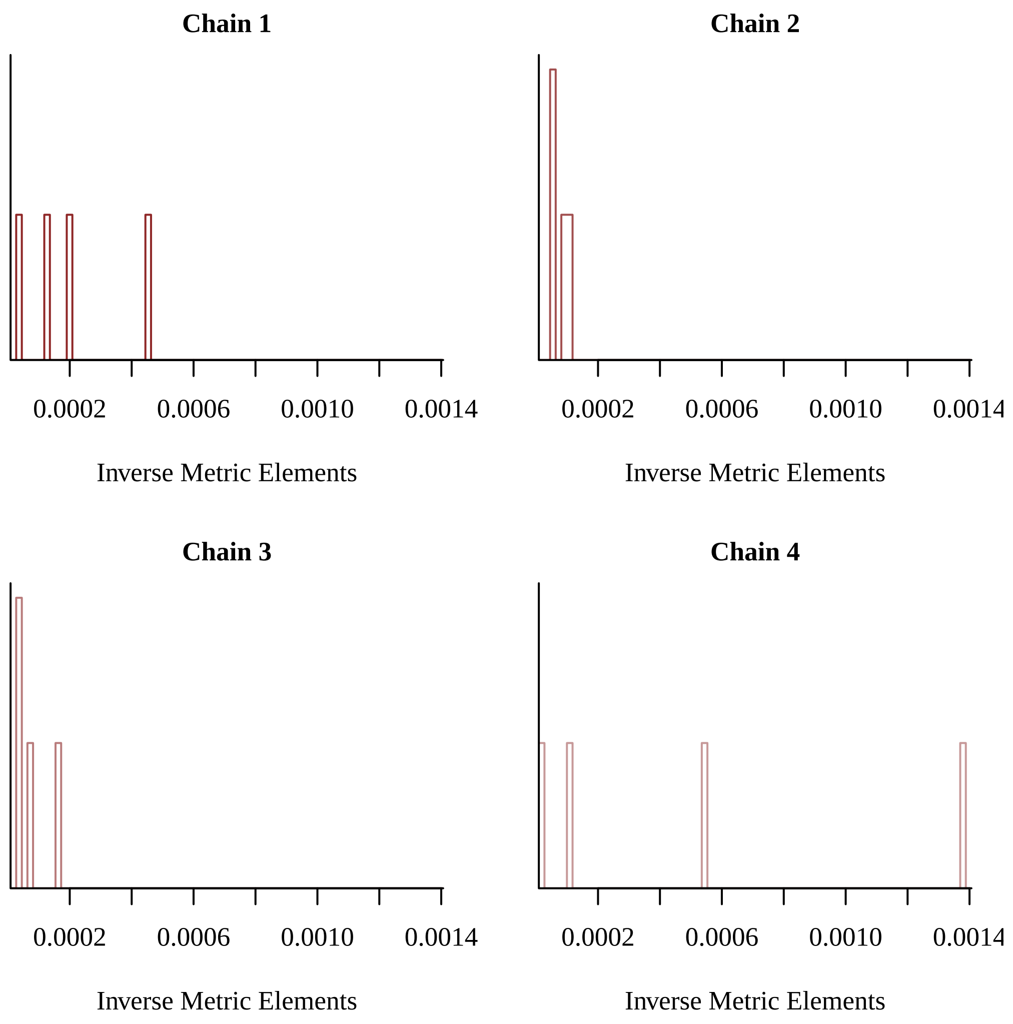
```
check_all_hmc_diagnostics(fit)
```

```
4064 of 4096 iterations ended with a divergence (99.21875%).
  Divergences are due unstable numerical integration.
  These instabilities are often due to posterior degeneracies.
  If there are only a small number of divergences then running
with adapt_delta larger than 0.801 may reduce the divergences
at the cost of more expensive transitions.

Chain 4: Average proxy acceptance statistic (0.629)
          is smaller than 90% of the target (0.801).
  A small average proxy acceptance statistic indicates that the
integrator step size adaptation failed to converge.  This is often
due to discontinuous or inexact gradients.
```

Almost every transition across the four Markov chains resulted in a divergence. This is due to the discontinuity in the linear probability model as the sudden jump from a finite to a negative infinite target density results in unstable numerical trajectories.

We also see the one of the Markov chains wasn't able to hit the step size adaptation target. To see why let's dig into the adapted configuration of the Hamiltonian Markov transition.

```
plot_inv_metric(fit, 75)
```



The problematic Markov chain also exhibits the most variation in its inverse metric elements, which in this case is probably an artifact of its warmup phase spending too much time close to a constraint boundary. Artificially variable inverse metric elements frustrate numerical integration which can then frustrate the integrator step size adaptation.

Interestingly the adapted step sizes are nearly the same for all four Markov chains. The lower average proxy acceptance statistic seen in the fourth Markov chain is due entirely to the wonky inverse metric adaptation.

```
display_stepsizes(fit)
```

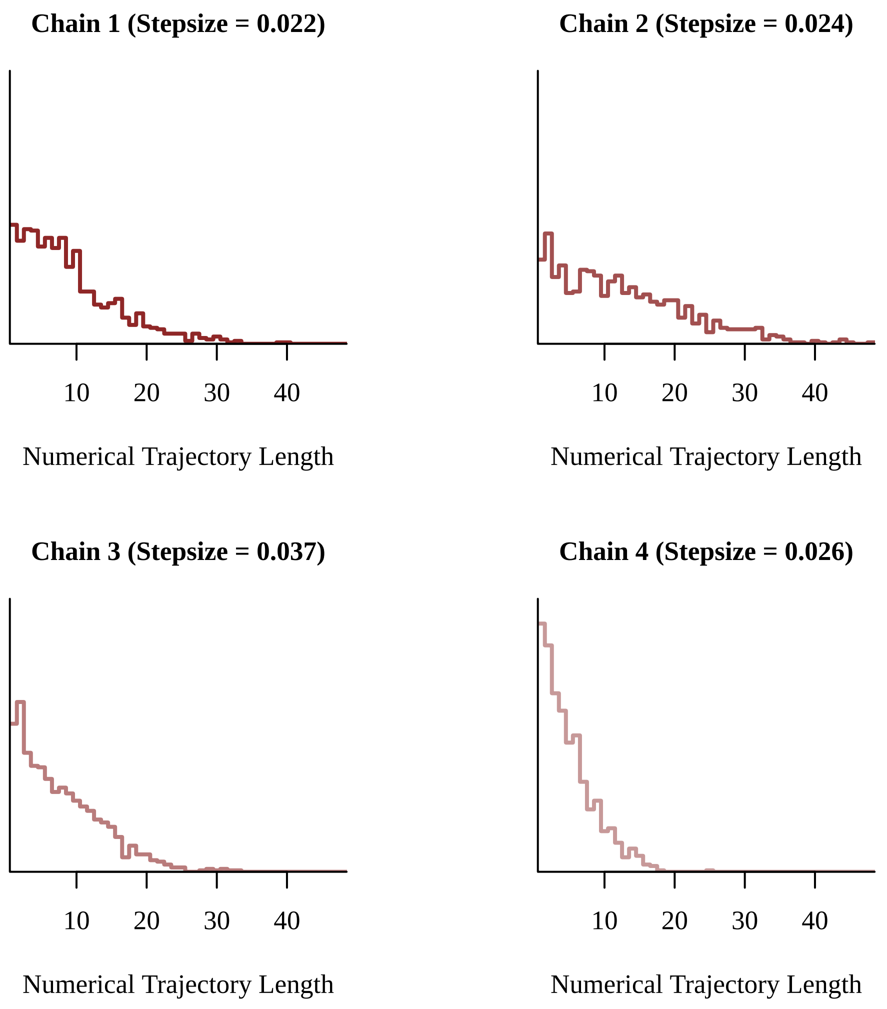
```
Chain 1: Integrator Step Size = 0.022103
Chain 2: Integrator Step Size = 0.024148
Chain 3: Integrator Step Size = 0.037035
Chain 4: Integrator Step Size = 0.026320
```

```
display_ave_accept_proxy(fit)
```

```
Chain 1: Average proxy acceptance statistic = 0.766
Chain 2: Average proxy acceptance statistic = 0.814
Chain 3: Average proxy acceptance statistic = 0.729
Chain 4: Average proxy acceptance statistic = 0.629
```

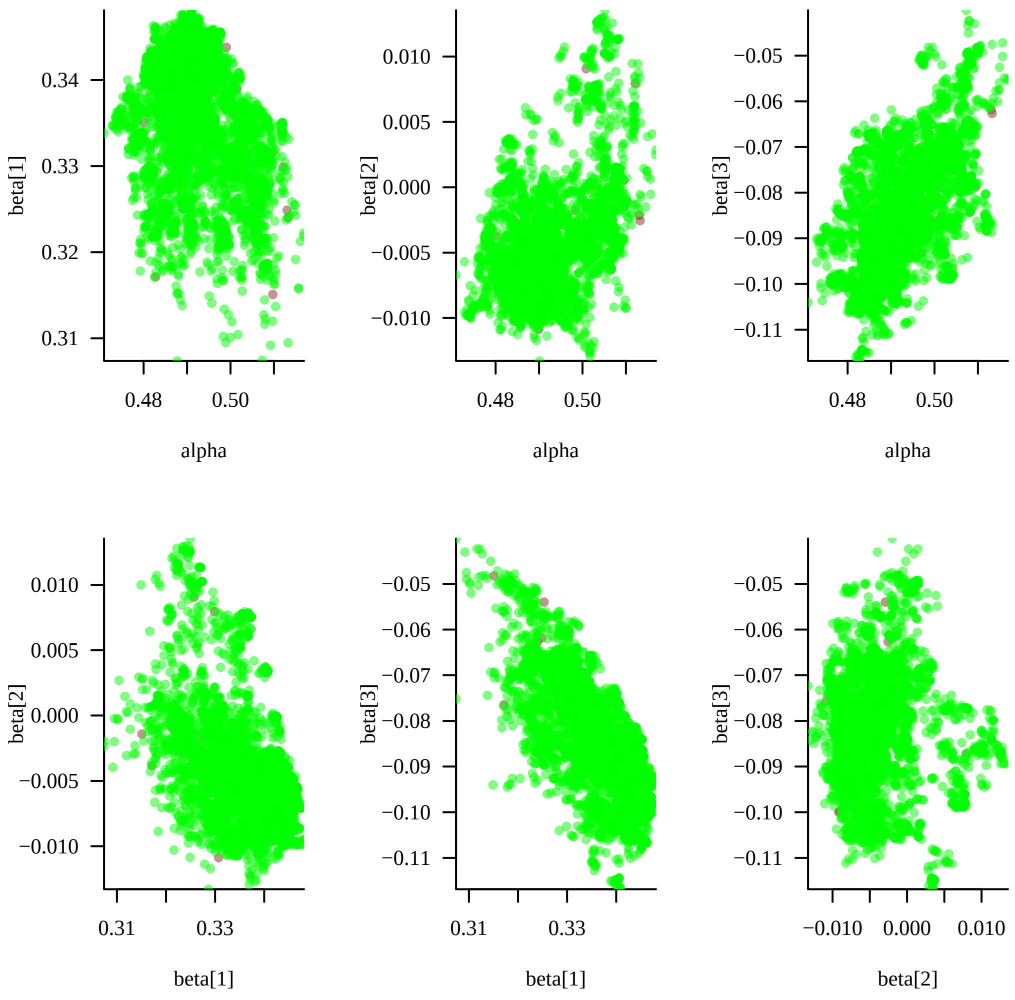
The different inverse metric results in different Hamiltonian dynamics. In this case the dynamics driving the fourth Markov chain are not able to explore as far as those in the other chains.

```
plot_num_leapfrog(fit)
```



Finally because nearly every transition is divergent we can't extract much information from the divergent-labeled pairs plots.

```
plot_div_pairs(fit, c("alpha", "beta[1]", "beta[2]", "beta[3]"),
               c(0, 0, 0, 0))
```



Having examined the Hamiltonian Monte Carlo diagnostics let's now look through the expectand specific diagnostics. By default we'll look at the parameter projection functions as well as all of the expectands defined in the `generated quantities` block.

Because of the Hamiltonian Monte Carlo diagnostic failures I'm going to limit the output just in case we have many failures for these diagnostics as well.

```
expectand_diagnostics_summary(fit)
```

```
The expectands 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22

The expectands 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22
  Split hat{R} larger than 1.1 is inconsistent with equilibrium.
```

```
The expectands 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22
  If hat{ESS} is too small then Markov chain Monte Carlo estimators will be too imprecise.
```

```
clip_output <- function(output, head, tail) {
  for(l in 1:head)
    cat(paste0(output[l], "\n"))
  cat("\n")
  cat("..........\n")
  cat("..........\n")
  cat("..........\n")
  cat("\n")
  N <- length(output)
  for(l in (N - tail):N)
    cat(paste0(output[l], "\n"))
}
```

```
clip_output(capture.output(check_all_expectand_diagnostics(fit)), 27, 23)
```

```
alpha:
  Split hat{R} (1.575) exceeds 1.1!
  Chain 1: hat{ESS} (16.304) is smaller than desired (100)!
  Chain 2: hat{ESS} (16.147) is smaller than desired (100)!
  Chain 3: hat{ESS} (7.943) is smaller than desired (100)!
  Chain 4: hat{ESS} (6.556) is smaller than desired (100)!

beta[1]:
  Split hat{R} (1.428) exceeds 1.1!
  Chain 1: hat{ESS} (6.385) is smaller than desired (100)!
  Chain 2: hat{ESS} (10.338) is smaller than desired (100)!
  Chain 3: hat{ESS} (11.444) is smaller than desired (100)!
  Chain 4: hat{ESS} (12.705) is smaller than desired (100)!

beta[2]:
  Split hat{R} (1.420) exceeds 1.1!
  Chain 1: hat{ESS} (12.499) is smaller than desired (100)!
  Chain 2: hat{ESS} (26.170) is smaller than desired (100)!
  Chain 3: hat{ESS} (5.394) is smaller than desired (100)!
  Chain 4: hat{ESS} (5.913) is smaller than desired (100)!
```

```
beta[3]:
  Split hat{R} (1.679) exceeds 1.1!
  Chain 1: hat{ESS} (7.130) is smaller than desired (100)!
  Chain 2: hat{ESS} (5.677) is smaller than desired (100)!
  Chain 3: hat{ESS} (5.136) is smaller than desired (100)!
  Chain 4: hat{ESS} (13.012) is smaller than desired (100)!

..........
..........
..........

p[999]:
  Split hat{R} (1.337) exceeds 1.1!
  Chain 1: hat{ESS} (14.811) is smaller than desired (100)!
  Chain 2: hat{ESS} (15.952) is smaller than desired (100)!
  Chain 3: hat{ESS} (6.228) is smaller than desired (100)!
  Chain 4: hat{ESS} (10.401) is smaller than desired (100)!

p[1000]:
  Split hat{R} (1.558) exceeds 1.1!
  Chain 1: hat{ESS} (15.340) is smaller than desired (100)!
  Chain 2: hat{ESS} (25.118) is smaller than desired (100)!
  Chain 3: hat{ESS} (4.992) is smaller than desired (100)!
  Chain 4: hat{ESS} (5.750) is smaller than desired (100)!

lp__:
  Chain 1: hat{ESS} (11.120) is smaller than desired (100)!
  Chain 2: hat{ESS} (54.510) is smaller than desired (100)!
  Chain 3: hat{ESS} (17.626) is smaller than desired (100)!
  Chain 4: hat{ESS} (69.311) is smaller than desired (100)!

Split hat{R} larger than 1.1 is inconsisent with equilibrium.

If hat{ESS} is too small then Markov chain Monte Carlo estimators will be too imprecise.
```

Well that output restriction proved to be prescient as most of the expectands are encountering problems. To avoid overwhelming ourselves let's focus on the four parameter expectands.

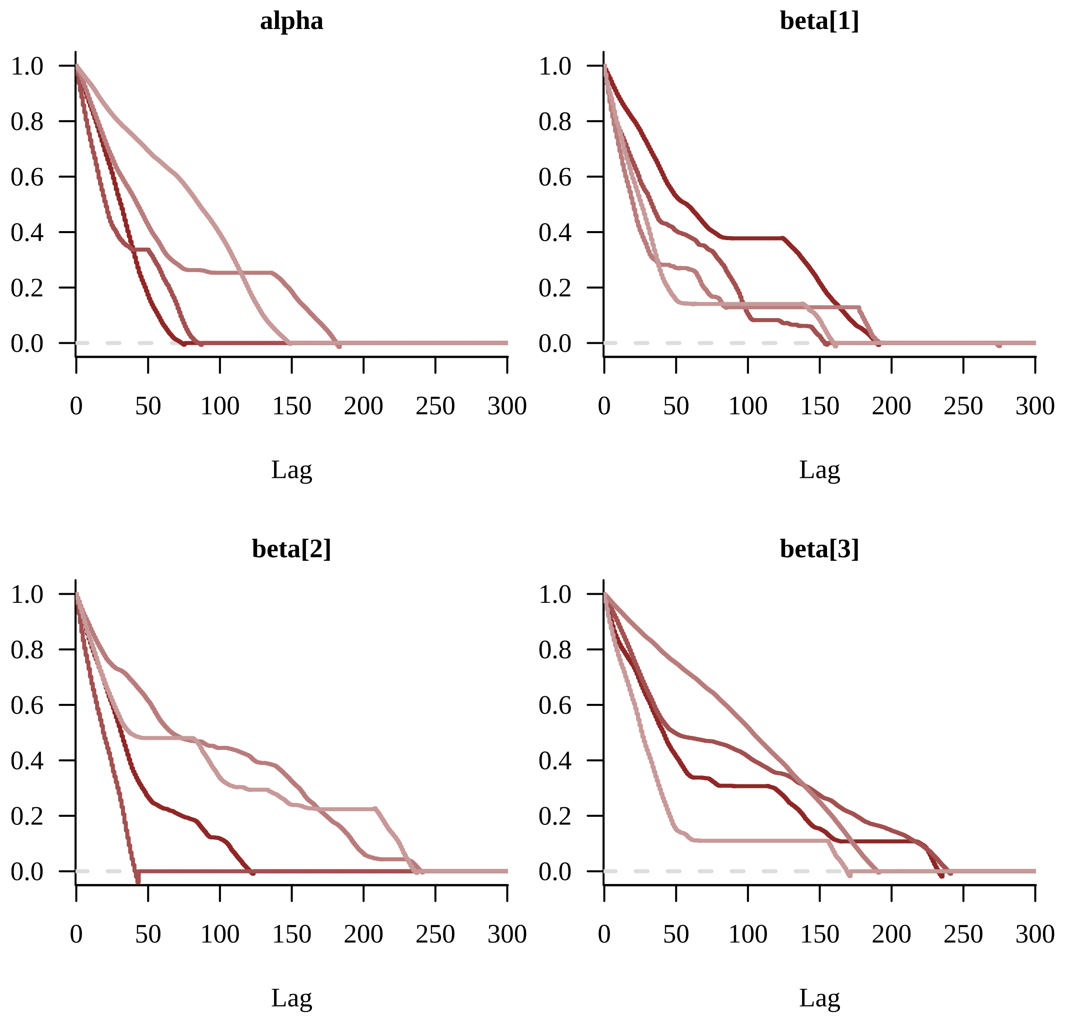
```r
check_all_expectand_diagnostics(fit, 1:4)
```

```
alpha:
  Split hat{R} (1.575) exceeds 1.1!
  Chain 1: hat{ESS} (16.304) is smaller than desired (100)!
  Chain 2: hat{ESS} (16.147) is smaller than desired (100)!
  Chain 3: hat{ESS} (7.943) is smaller than desired (100)!
  Chain 4: hat{ESS} (6.556) is smaller than desired (100)!

beta[1]:
  Split hat{R} (1.428) exceeds 1.1!
  Chain 1: hat{ESS} (6.385) is smaller than desired (100)!
  Chain 2: hat{ESS} (10.338) is smaller than desired (100)!
  Chain 3: hat{ESS} (11.444) is smaller than desired (100)!
  Chain 4: hat{ESS} (12.705) is smaller than desired (100)!

beta[2]:
  Split hat{R} (1.420) exceeds 1.1!
  Chain 1: hat{ESS} (12.499) is smaller than desired (100)!
  Chain 2: hat{ESS} (26.170) is smaller than desired (100)!
  Chain 3: hat{ESS} (5.394) is smaller than desired (100)!
  Chain 4: hat{ESS} (5.913) is smaller than desired (100)!

beta[3]:
  Split hat{R} (1.679) exceeds 1.1!
  Chain 1: hat{ESS} (7.130) is smaller than desired (100)!
  Chain 2: hat{ESS} (5.677) is smaller than desired (100)!
  Chain 3: hat{ESS} (5.136) is smaller than desired (100)!
  Chain 4: hat{ESS} (13.012) is smaller than desired (100)!

Split hat{R} larger than 1.1 is inconsisent with equilibrium.

If hat{ESS} is too small then Markov chain Monte Carlo estimators will be too imprecise.
```

All four parameter expectands exhibit split $\hat{R}$ warnings and low empirical effective sample size warnings. The question is whether or not the split $\hat{R}$ warnings indicate quasistationarity or just insufficient exploration.

Motivated by the small effective sample size estimates let's look at the empirical correlograms for each parameter expectand.

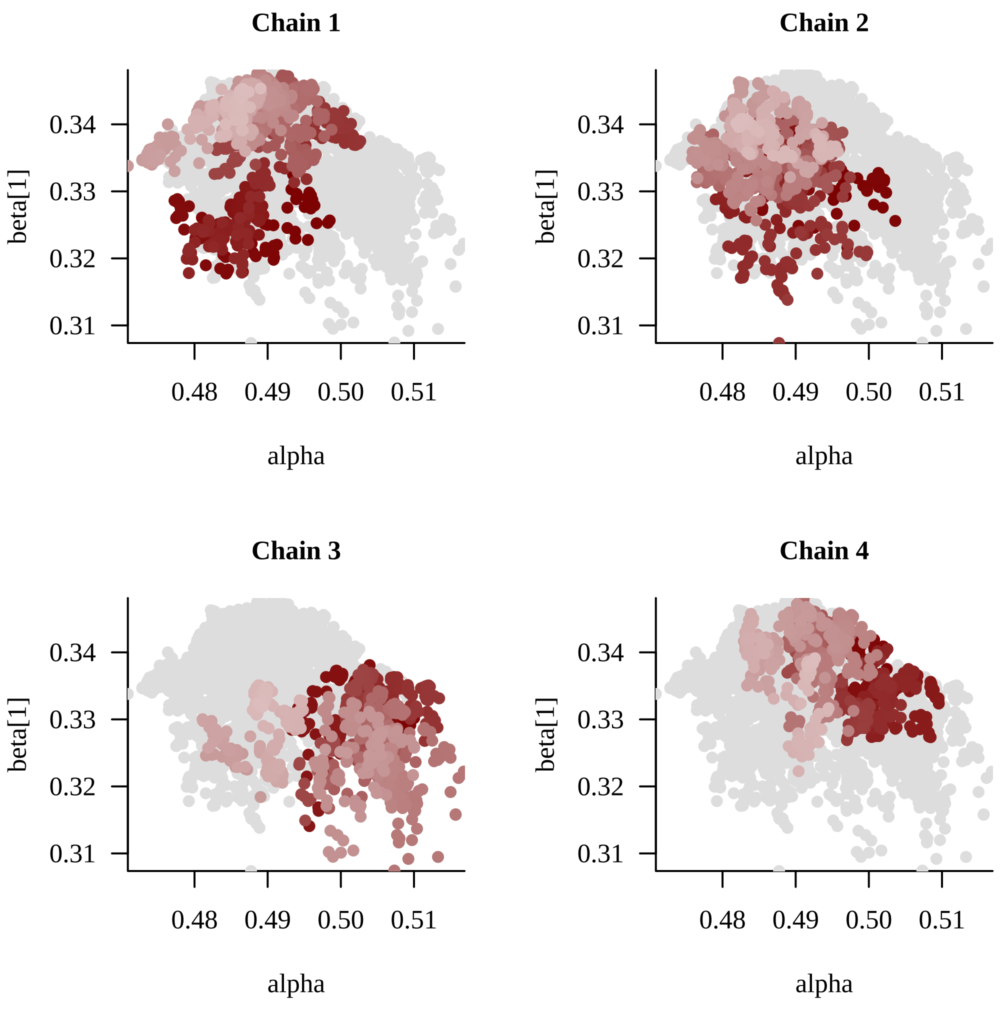
```r
unpermuted_samples <- rstan:::extract(fit, permute=FALSE)

par(mfrow=c(2, 2), mar = c(5, 2, 2, 1))
plot_empirical_correlogram(unpermuted_samples[,,1], 300,
                               rholim=c(-0.05, 1.05), "alpha")
plot_empirical_correlogram(unpermuted_samples[,,2], 300,
                               rholim=c(-0.05, 1.05), "beta[1]")
plot_empirical_correlogram(unpermuted_samples[,,3], 300,
                               rholim=c(-0.05, 1.05), "beta[2]")
plot_empirical_correlogram(unpermuted_samples[,,4], 300,
                               rholim=c(-0.05, 1.05), "beta[3]")
```



Regardless of whether or not these Markov chains are stationary they are extremely autocorrelated. Assuming stationarity we don't start to forget the beginning of each Markov chain until we've worked through a quarter of the total length, leaving only about four independent samples across each chain.

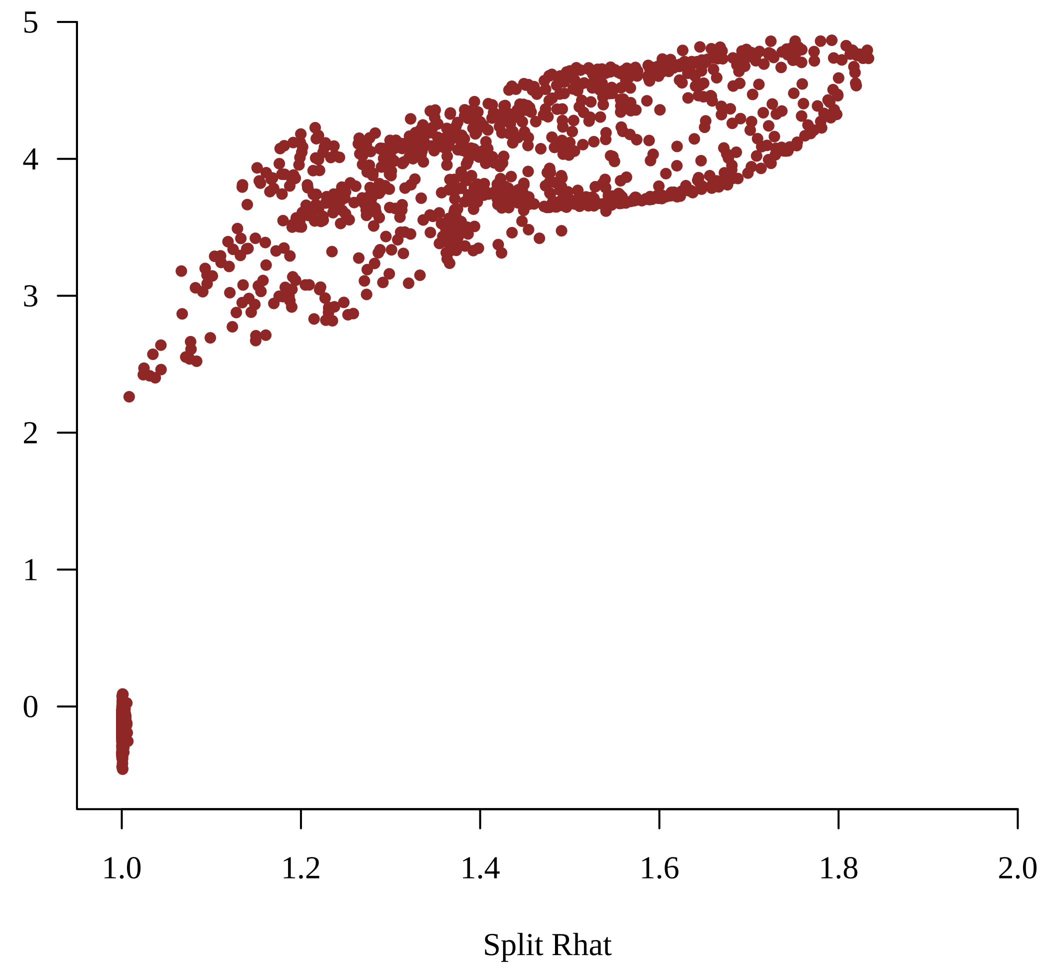
This is consistent with the constraint violations breaking the coherent, gradient-driven exploration of Hamiltonian Monte Carlo so that the Markov chains devolve into diffuse random walks. Indeed looking at the chain-separated pairs plots we see the spatial color continuity characteristic of a random walk.

```
plot_chain_sep_pairs(unpermuted_samples[,,1], "alpha", unpermuted_samples[,,2], "beta[1]")
```



To more quantitatively blame the large split $\hat{R}$s on these strong autocorrelations we can plot the split $\hat{R}$ from each expectand against the corresponding empirical integrated autocorrelation time across. Specifically for each expectand we plot split $\hat{R}$ against we use the smallest empirical integrated autocorrelation of the four Markov chains.

```r
rhats <- compute_split_rhats(fit)
min_int_ac_times <- compute_min_int_ac_times(fit)

par(mfrow=c(1, 1), mar = c(5, 2, 2, 1))
plot(rhats, log(min_int_ac_times),
     col=c_dark, pch=16, cex=0.8,
     xlab="Split Rhat", xlim=c(0.95, 2),
     ylab="Minimum Integrated Autocorrelation Time", ylim=c(-0.75, 5))
```



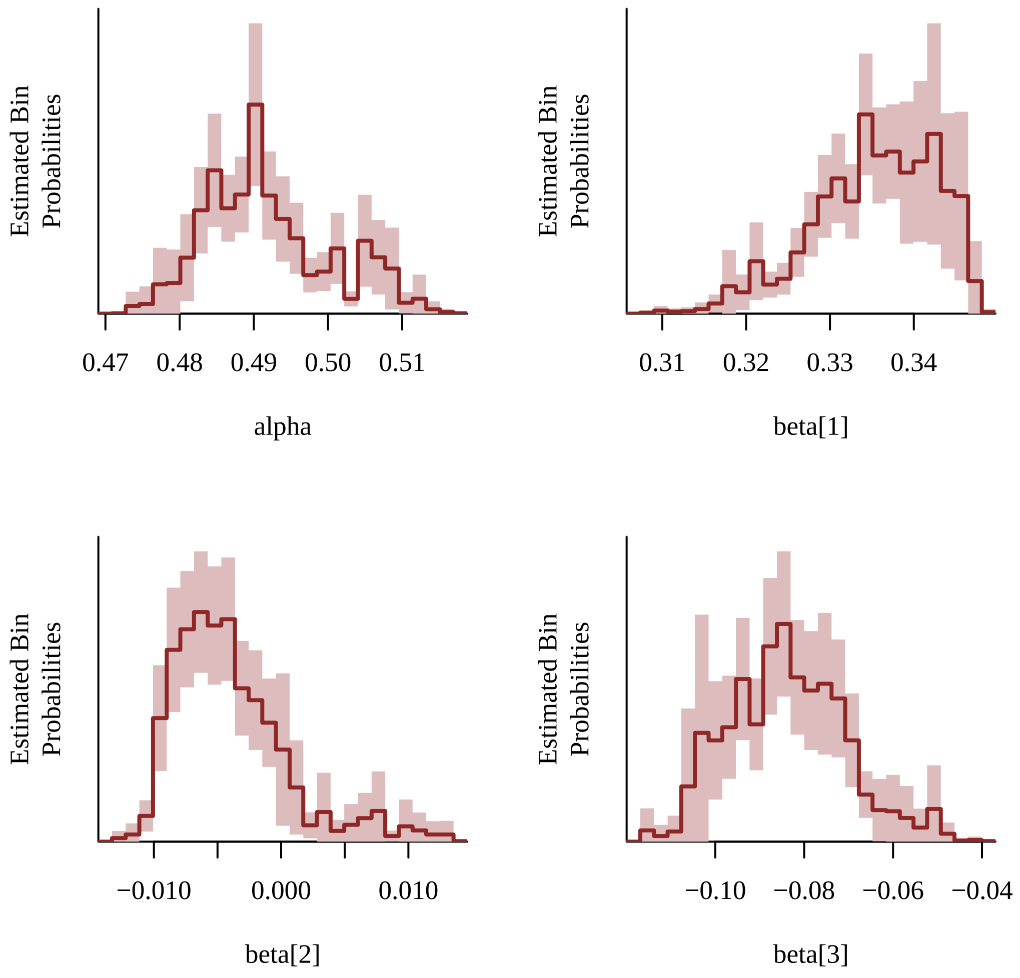
Every expectand with a large split $\hat{R}$s also exhibits a large value the minimum empirical integrated autocorrelation time, confirming that the latter are due to our Markov chains not containing enough information.

If we are sloppy, ignore these diagnostics, and assume that all of our Markov chain Monte

Carlo estimators are accurate then we are quickly mislead about the actual behavior of the posterior distribution. One way to guard against this sloppiness is to always accompany a Markov chain Monte Carlo estimator with an estimated error. Even if that error is inaccurate it can sometimes communicate underlying problems.

For example let's look at a pushforward histogram for each parameter with light red bands visualizing the standard error around the bin probability estimates in dark red.

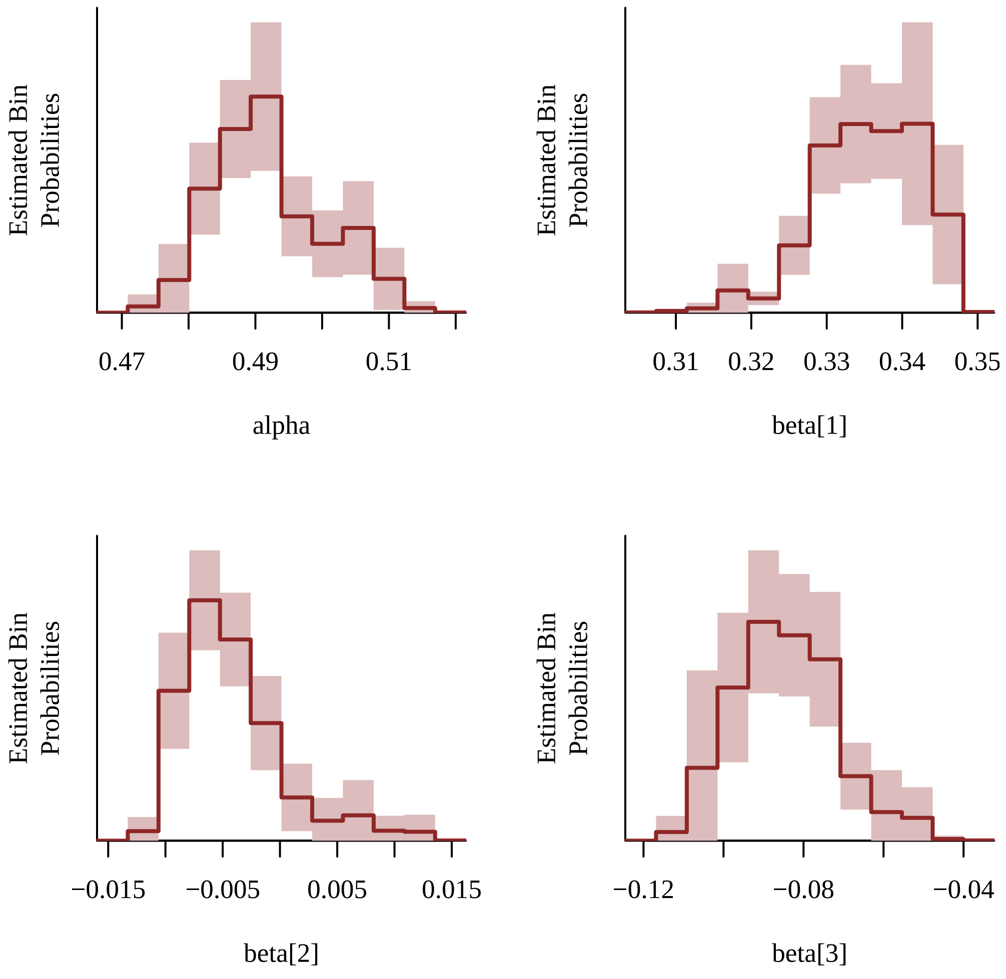
```
par(mfrow=c(2, 2), mar = c(5, 4, 2, 1))
plot_pushforward_hist(unpermuted_samples[,,1], 25, name="alpha")
plot_pushforward_hist(unpermuted_samples[,,2], 25, name="beta[1]")
plot_pushforward_hist(unpermuted_samples[,,3], 25, name="beta[2]")
plot_pushforward_hist(unpermuted_samples[,,4], 25, name="beta[3]")
```



If we look at the central estimates alone we might convince ourselves of all kinds of interesting structure. For example potential multi-modality in `alpha` and `beta[2]` and platykurticity in `beta[1]` and `beta[3]`. These structures, however, are all within the scope of the relatively large standard error bands which suggests that they are all consistent with estimator noise.

Reducing the number of bins decreases the relative standard errors but at the same time many of the visual artifacts recede.

```
par(mfrow=c(2, 2), mar = c(5, 4, 2, 1))
plot_pushforward_hist(unpermuted_samples[,,1], 10, name="alpha")
plot_pushforward_hist(unpermuted_samples[,,2], 10, name="beta[1]")
plot_pushforward_hist(unpermuted_samples[,,3], 10, name="beta[2]")
plot_pushforward_hist(unpermuted_samples[,,4], 10, name="beta[3]")
```



When the bin indicator functions enjoy Markov chain Monte Carlo central limit theorems these standard error bands allow us to discriminate between meaningful structure and accidental artifacts regardless of the histogram binning. Even if central limit theorems don't hold the error bands provide one more way that we can potentially diagnose untrustworthy computation.

License

Original Computing Environment

```
writeLines(readLines(file.path(Sys.getenv("HOME"), ".R/Makevars")))
```

```
CC=clang

CXXFLAGS=-O3 -mtune=native -march=native -Wno-unused-variable -Wno-unused-function -Wno-macro
CXX=clang++ -arch x86_64 -ftemplate-depth-256

CXX14FLAGS=-O3 -mtune=native -march=native -Wno-unused-variable -Wno-unused-function -Wno-mac
CXX14=clang++ -arch x86_64 -ftemplate-depth-256
```

```
sessionInfo()
```

```
R version 4.0.2 (2020-06-22)
Platform: x86_64-apple-darwin17.0 (64-bit)
Running under: macOS Catalina 10.15.7

Matrix products: default
BLAS:   /Library/Frameworks/R.framework/Versions/4.0/Resources/lib/libRblas.dylib
LAPACK: /Library/Frameworks/R.framework/Versions/4.0/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     datasets  methods   base

other attached packages:
[1] rstan_2.19.3        ggplot2_3.3.1       StanHeaders_2.21.0-3
[4] colormap_0.1.4

loaded via a namespace (and not attached):
 [1] Rcpp_1.0.4.6       compiler_4.0.2     pillar_1.4.4       prettyunits_1.1.1
 [5] tools_4.0.2        pkgbuild_1.0.8     digest_0.6.25      jsonlite_1.6.1
 [9] evaluate_0.17      lifecycle_0.2.0    tibble_3.0.1       gtable_0.3.0
[13] pkgconfig_2.0.3    rlang_0.4.6        cli_2.0.2          parallel_4.0.2
[17] curl_4.3           yaml_2.2.1         xfun_0.33          loo_2.2.0
[21] gridExtra_2.3      withr_2.2.0        stringr_1.4.0      dplyr_1.0.0
[25] knitr_1.40         generics_0.0.2     vctrs_0.3.0        stats4_4.0.2
[29] grid_4.0.2         tidyselect_1.1.0   inline_0.3.15      glue_1.4.1
[33] R6_2.4.1           processx_3.4.2     fansi_0.4.1        rmarkdown_2.2
[37] callr_3.4.3        purrr_0.3.4        magrittr_1.5       codetools_0.2-16
[41] matrixStats_0.56.0 ps_1.3.3           scales_1.1.1       htmltools_0.4.0
[45] ellipsis_0.3.1     assertthat_0.2.1   colorspace_1.4-1   V8_3.2.0
[49] stringi_1.4.6      munsell_0.5.0      crayon_1.3.4
```