Markov Chain Monte Carlo Diagnostics

Michael Betancourt

11/1/22

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.

```
import matplotlib
import matplotlib.pyplot as plot
plot.rcParams['figure.figsize'] = [6, 4]
plot.rcParams['figure.dpi'] = 100
plot.rcParams['font.family'] = "Serif"
from matplotlib.colors import LinearSegmentedColormap

light="#DCBCBC"
light_highlight="#C79999"
mid="#B97C7C"
mid_highlight="#A25050"
dark="#8F2727"
dark_highlight="#7C0000"
```

At the same time let's introduce a helper function to cache Stan executables.

```
import multiprocessing
multiprocessing.set_start_method("fork")

import pystan
import pickle

def compile_model(filename, model_name=None, **kwargs):
  """This will automatically cache models - great if you're just running a
     script on the command line.

    See http://pystan.readthedocs.io/en/latest/avoiding_recompilation.html"""
  from hashlib import md5

  with open(filename) as f:
    model_code = f.read()
    code_hash = md5(model_code.encode('ascii')).hexdigest()
    if model_name is None:
      cache_fn = 'cached-model-{}.pkl'.format(code_hash)
    else:
```

```
    cache_fn = 'cached-{}-{}.pkl'.format(model_name, code_hash)
  try:
    sm = pickle.load(open(cache_fn, 'rb'))
  except:
    sm = pystan.StanModel(model_code=model_code)
    with open(cache_fn, 'wb') as f:
      pickle.dump(sm, f)
  else:
     print("Using cached StanModel")
  return sm
```

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.

```
import numpy

def check_all_hmc_diagnostics(fit,
                              adapt_target=0.801,
                              max_treedepth=10):
  """Check all Hamiltonian Monte Carlo Diagnostics for an
     ensemble of Markov chains"""
  sampler_params = fit.get_sampler_params(inc_warmup=False)

  no_warning = True

  # Check divergences
  divergent = [x for y in sampler_params for x in y['divergent__']]
  n = sum(divergent)
  N = len(divergent)

  if n > 0:
    no_warning = False
    print(f'{n:.0f} of {N} iterations ended with a divergence ({n / N:.2%})')
    print(   '  Divergences are due unstable numerical integration.\n'
```

```
        +  '  These instabilities are often due to posterior degeneracies.\n'
        +  '  If there are only a small number of divergences then running\n'
        + f'with adapt_delta larger than {adapt_target:.3f} may reduce the\n'
        + 'divergences at the cost of more expensive transitions.\n')

# Check transitions that ended prematurely due to maximum tree depth limit
sampler_params = fit.get_sampler_params(inc_warmup=False)
depths = [x for y in sampler_params for x in y['treedepth__']]
n = sum(1 for x in depths if x == max_treedepth)
N = len(depths)

if n > 0:
  no_warning = False
  print(  f'{n:.0f} of {N} iterations saturated the maximum tree depth of '
        + f'{max_treedepth} ({n / N:.2%})')
  print(  '  Increasing max_depth will increase the efficiency of the '
        + 'transitions.\n')

# Checks the energy fraction of missing information (E-FMI)
no_efmi_warning = True
for chain_num, s in enumerate(sampler_params):
  energies = s['energy__']
  numer = sum((energies[i] - energies[i - 1])**2 for i in range(1, len(energies))) / len
  denom = numpy.var(energies)
  if numer / denom < 0.2:
    no_warning = False
    no_efmi_warning = False
    print(f'Chain {chain_num + 1}: E-FMI = {numer / denom:.3f}.')

if not no_efmi_warning:
  print('  E-FMI below 0.2 suggests a funnel-like geometry hiding')
  print('somewhere in the posterior distribution.\n')

# Check convergence of the stepsize adaptation
no_accept_warning = True
for chain_num, s in enumerate(sampler_params):
  ave_accept_proxy = numpy.mean(s['accept_stat__'])
  if ave_accept_proxy < 0.9 * adapt_target:
    no_warning = False
    no_accept_warning = False
    print(  f'Chain {chain_num + 1}: Average proxy acceptance statistic '
```

```
            + f'({ave_accept_proxy:.3f})')
      print(  f'         is smaller than 90% of the target '
            + f'({adapt_target:.3f})')

  if not no_accept_warning:
    print('  A small average proxy acceptance statistic indicates that the')
    print('integrator step size adaptation failed to converge.  This is often')
    print('due to discontinuous or inexact gradients.\n\n')

  if no_warning:
    print('All Hamiltonian Monte Carlo diagnostics are consistent with')
    print('accurate Markov chain Monte Carlo.\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.

```
import math
import re

def plot_inv_metric(fit, B=25):
  """Plot outcome of inverse metric adaptation"""
  chain_info = fit.get_adaptation_info()
  C = len(chain_info)

  inv_metric_elems = [None] * C
  for c, raw_info in enumerate(chain_info):
    clean1 = re.sub("# Adaptation terminated\n# Step size = [0-9.]*\n#",
                    "", raw_info)
    clean2 = re.sub(" [a-zA-Z ]*:\n# ", "", clean1)
    clean3 = re.sub("\n$", "", clean2)
    inv_metric_elems[c] = [float(s) for s in clean3.split(',')]

  min_elem = min([ min(a) for a in inv_metric_elems ])
```

```
max_elem = max([ max(a) for a in inv_metric_elems ])

delta = (max_elem - min_elem) / B
min_elem = min_elem - delta
max_elem = max_elem + delta
bins = numpy.arange(min_elem, max_elem + delta, delta)
B = B + 2

max_y = max([ max(numpy.histogram(a, bins=bins)[0]) for a in inv_metric_elems ])

idxs = [ idx for idx in range(B) for r in range(2) ]
xs = [ bins[idx + delta] for idx in range(B) for delta in [0, 1]]

N_plots = C
N_cols = 2
N_rows = math.ceil(N_plots / N_cols)
f, axarr = plot.subplots(N_rows, N_cols)
k = 0

sampler_params = fit.get_sampler_params(inc_warmup=False)
sci_formatter = matplotlib.ticker.FuncFormatter(lambda x, lim: f'{x:.1e}')

for c in range(C):
  counts = numpy.histogram(inv_metric_elems[c], bins=bins)[0]
  ys = counts[idxs]

  eps = sampler_params[c]['stepsize__'][0]

  idx1 = k // N_cols
  idx2 = k % N_cols
  k += 1

  axarr[idx1, idx2].plot(xs, ys, dark)
  axarr[idx1, idx2].set_title(f'Chain {c + 1}\n(Stepsize = {eps:.3e})')
  axarr[idx1, idx2].set_xlabel("Inverse Metric Elements")
  axarr[idx1, idx2].set_xlim([min_elem, max_elem])
  axarr[idx1, idx2].get_xaxis().set_major_formatter(sci_formatter)
  axarr[idx1, idx2].set_ylabel("")
  axarr[idx1, idx2].get_yaxis().set_visible(False)
  axarr[idx1, idx2].set_ylim([0, 1.05 * max_y])
  axarr[idx1, idx2].spines["top"].set_visible(False)
```

```
    axarr[idx1, idx2].spines["left"].set_visible(False)
    axarr[idx1, idx2].spines["right"].set_visible(False)

  plot.subplots_adjust(hspace=1.0, wspace=0.25)
  plot.show()
```

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.

```
def display_stepsizes(fit):
  """Display outcome of symplectic integrator step size adaptation"""
  sampler_params = fit.get_sampler_params(inc_warmup=False)
  for chain_num, s in enumerate(sampler_params):
    stepsize = s['stepsize__'][0]
    print(f'Chain {chain_num + 1}: Integrator Step Size = {stepsize:.2e}')
```

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.

```
def plot_num_leapfrog(fit):
  """Display symplectic integrator trajectory lenghts"""
  sampler_params = fit.get_sampler_params(inc_warmup=False)

  vals_counts = [ numpy.unique(s['n_leapfrog__'], return_counts=True)
                  for s in sampler_params ]
  max_n = max([ max(a[0]) for a in vals_counts ]).astype(numpy.int64)
  max_counts = max([ max(a[1]) for a in vals_counts ])

  idxs = [ idx for idx in range(max_n) for r in range(2) ]
  xs = [ idx + delta for idx in range(max_n) for delta in [-0.5, 0.5]]

  C = len(vals_counts)
```

```
  N_plots = C
  N_cols = 2
  N_rows = math.ceil(N_plots / N_cols)
  f, axarr = plot.subplots(N_rows, N_cols)
  k = 0

  for c, s in enumerate(sampler_params):
    counts = numpy.histogram(s['n_leapfrog__'],
                             bins=numpy.arange(0.5, max_n + 1.5, 1))[0]
    ys = counts[idxs]

    eps = s['stepsize__'][0]

    idx1 = k // N_cols
    idx2 = k % N_cols
    k += 1

    axarr[idx1, idx2].plot(xs, ys, dark)
    axarr[idx1, idx2].set_title(f'Chain {c + 1}\n(Stepsize = {eps:.3e})')
    axarr[idx1, idx2].set_xlabel("Numerical Trajectory Length")
    axarr[idx1, idx2].set_xlim([0.5, max_n + 0.5])
    axarr[idx1, idx2].set_ylabel("")
    axarr[idx1, idx2].get_yaxis().set_visible(False)
    axarr[idx1, idx2].set_ylim([0, 1.1 * max_counts])
    axarr[idx1, idx2].spines["top"].set_visible(False)
    axarr[idx1, idx2].spines["right"].set_visible(False)

  plot.subplots_adjust(hspace=1.0, wspace=0.25)
  plot.show()
```

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.

```
def display_ave_accept_proxy(fit):
  """Display empirical average of the proxy acceptance statistic
     across each individual Markov chains"""
  sampler_params = fit.get_sampler_params(inc_warmup=False)
```

```
  for c, s in enumerate(sampler_params):
    ave_accept_proxy = numpy.mean(s['accept_stat__'])
    print(  f'Chain {c + 1}: '
          + f'Average proxy acceptance statistic = {ave_accept_proxy:.3f}')
```

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.

```
def _by_chain(unpermuted_extraction):
  num_chains = len(unpermuted_extraction[0])
  result = [[] for _ in range(num_chains)]
  for c in range(num_chains):
    for i in range(len(unpermuted_extraction)):
      result[c].append(unpermuted_extraction[i][c])
  return numpy.array(result)

def _shaped_ordered_params(fit):
  # flattened, unpermuted, by (iteration, chain)
  ef = fit.extract(permuted=False, inc_warmup=False)
  ef = _by_chain(ef)
  ef = ef.reshape(-1, len(ef[0][0]))
  ef = ef[:, 0:len(fit.flatnames)] # drop lp__
  shaped = {}
  idx = 0
  for dim, param_name in zip(fit.par_dims, fit.extract().keys()):
    length = int(numpy.prod(dim))
    shaped[param_name] = ef[:,idx:idx + length]
    shaped[param_name].reshape(*([-1] + dim))
    idx += length
  return shaped

def partition_div(fit):
```

```
  """Separate Markov chain states into those sampled from non-divergent
     numerical Hamiltonian trajectories and those sampled from divergent
     numerical Hamiltonian trajectories"""
  sampler_params = fit.get_sampler_params(inc_warmup=False)
  div = numpy.concatenate([x['divergent__'] for x in
                           sampler_params]).astype('int')
  params = _shaped_ordered_params(fit)
  nondiv_params = dict((key, params[key][div == 0]) for key in params)
  div_params = dict((key, params[key][div == 1]) for key in params)
  return nondiv_params, div_params
```

```
def plot_div_pairs(fit, names, transforms):
  """Plot pairwise scatter plots with non-divergent and divergent
     transitions separated by color"""
  nondiv_samples, div_samples = partition_div(fit)
  N_nondiv = len(nondiv_samples[list(nondiv_samples.keys())[0]])
  N_div = len(div_samples[list(div_samples.keys())[0]])

  N = len(names)
  N_plots = math.comb(N, 2)
  N_cols = 2
  N_rows = math.ceil(N_plots / N_cols)
  f, axarr = plot.subplots(N_rows, N_cols)

  k = 0

  for n in range(N - 1):
    for m in range(n + 1, N):

      name_x = names[n]

      if re.search('\[', name_x):
        base_name, idxs = name_x.split('[')
        index_name_x = base_name
        idxs = re.sub('\]', '', idxs)
        nondiv_idxs = tuple([slice(0, N_nondiv)]) + tuple([int(s) - 1 for s in idxs.split(
        div_idxs = tuple([slice(0, N_div)]) + tuple([int(s) - 1 for s in idxs.split(',')])
      else:
        index_name_x = name_x
        nondiv_idxs = tuple([slice(0, N_nondiv)]) + (0,)
        div_idxs = tuple([slice(0, N_div)]) + (0,)
```

```
if transforms[n] == 0:
  x_nondiv_samples = nondiv_samples[index_name_x][nondiv_idxs]
  x_div_samples = div_samples[index_name_x][div_idxs]
  x_name = name_x
elif transforms[n] == 1:
  x_nondiv_samples = [math.log(x) for x in nondiv_samples[index_name_x][nondiv_idxs]
  x_div_samples = [math.log(x) for x in div_samples[index_name_x][div_idxs]]
  x_name = "log(" + name_x + ")"
xmin = min(numpy.concatenate((x_nondiv_samples, x_div_samples)))
xmax = max(numpy.concatenate((x_nondiv_samples, x_div_samples)))

name_y = names[m]

if re.search('\[', name_y):
  base_name, idxs = name_y.split('[')
  index_name_y = base_name
  idxs = re.sub('\]', '', idxs)
  nondiv_idxs = tuple([slice(0, N_nondiv)]) + tuple([int(s) - 1 for s in idxs.split(
  div_idxs = tuple([slice(0, N_div)]) + tuple([int(s) - 1 for s in idxs.split(',')])
else:
  index_name_y = name_y
  nondiv_idxs = tuple([slice(0, N_nondiv)]) + (0,)
  div_idxs = tuple([slice(0, N_div)]) + (0,)

if transforms[m] == 0:
  y_nondiv_samples = nondiv_samples[index_name_y][nondiv_idxs]
  y_div_samples = div_samples[index_name_y][div_idxs]
  y_name = name_y
elif transforms[m] == 1:
  y_nondiv_samples = [math.log(y) for y in nondiv_samples[index_name_y][nondiv_idxs]
  y_div_samples = [math.log(y) for y in div_samples[index_name_y][div_idxs]]
  y_name = "log(" + name_y + ")"
ymin = min(numpy.concatenate((y_nondiv_samples, y_div_samples)))
ymax = max(numpy.concatenate((y_nondiv_samples, y_div_samples)))

idx1 = k // N_cols
idx2 = k % N_cols
k += 1

axarr[idx1, idx2].scatter(x_nondiv_samples, y_nondiv_samples, s=10,
                          color = dark_highlight, alpha=0.05)
```

```
    axarr[idx1, idx2].scatter(x_div_samples, y_div_samples, s=10,
                              color = "#00FF00", alpha=0.25)
    axarr[idx1, idx2].set_xlabel(x_name)
    axarr[idx1, idx2].set_xlim([xmin, xmax])
    axarr[idx1, idx2].set_ylabel(y_name)
    axarr[idx1, idx2].set_ylim([ymin, ymax])
    axarr[idx1, idx2].spines["top"].set_visible(False)
    axarr[idx1, idx2].spines["right"].set_visible(False)

for i in range(N_rows * N_cols - N_plots):
  idx1 = k // N_cols
  idx2 = k % N_cols
  k += 1
  axarr[idx1, idx2].axis('off')

plot.subplots_adjust(hspace=0.75, wspace=0.75)
plot.show()
```

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.

```
def compute_khat(fs):
  """Compute empirical Pareto shape for a positive sample"""
  N = len(fs)
  sorted_fs = sorted(fs)

  if sorted_fs[0] == sorted_fs[-1]:
    return -2

  if (sorted_fs[0] < 0):
    print("Sequence values must be positive!")
    return NaN

  # Estimate 25% quantile
  q = sorted_fs[math.floor(0.25 * N + 0.5)]
  if q == sorted_fs[0]:
    return -2

  # Heurstic Pareto configuration
  M = 20 + math.floor(math.sqrt(N))

  b_hat_vec = [None] * M
  log_w_vec = [None] * M

  for m in range(M):
    b_hat_vec[m] = 1 / sorted_fs[-1] + (1 - math.sqrt(M / (m + 0.5))) / (3 * q)
    if b_hat_vec[m] != 0:
      k_hat = - numpy.mean( [ math.log(1 - b_hat_vec[m] * f) for f in sorted_fs ] )
      log_w_vec[m] = N * ( math.log(b_hat_vec[m] / k_hat) + k_hat - 1)
    else:
      log_w_vec[m] = 0

  # Remove terms that don't contribute to improve numerical stability of average
```

```
  log_w_vec = [ lw for lw in log_w_vec if lw != 0 ]
  b_hat_vec = [ b for b in b_hat_vec if b != 0 ]

  max_log_w = max(log_w_vec)
  b_hat = sum( [ b * math.exp(lw - max_log_w) for b, lw in zip(b_hat_vec, log_w_vec) ] ) /
          sum( [ math.exp(lw - max_log_w) for lw in log_w_vec ] )

  return numpy.mean( [ math.log(1 - b_hat * f) for f in sorted_fs ] )
```

```
def compute_tail_khats(fs):
  """Compute empirical Pareto shape for upper and lower tails"""
  f_center = numpy.median(fs)
  fs_left = [ math.fabs(f - f_center) for f in fs if f <= f_center ]
  fs_right = [ f - f_center for f in fs if f > f_center ]

  # Default to -2 if left tail is ill-defined
  khat_left = -2
  if len(fs_left) > 40:
    khat_left = compute_khat(fs_left)

  # Default to -2 if right tail is ill-defined
  khat_right = -2
  if len(fs_right) > 40:
    khat_right = compute_khat(fs_right)

  return [khat_left, khat_right]
```

```
def check_tail_khats(unpermuted_samples):
  """Check empirical Pareto shape for upper and lower tails of a
     given expectand output ensemble"""
  no_warning = True
  C = len(unpermuted_samples[0,:])
  for c in range(C):
    fs = unpermuted_samples[:,c]
    khats = compute_tail_khats(fs)
    if khats[0] >= 0.25 and khats[1] >= 0.25:
      no_warning = False
      print(f'Chain {c + 1}: Both left and right tail khats exceed 0.25!')
    elif khats[0] < 0.25 and khats[1] >= 0.25:
      no_warning = False
```

```
      print(f'Chain {c + 1}: Right tail khat exceeds 0.25!')
    elif khats[0] >= 0.25 and khats[1] < 0.25:
      no_warning = False
      print(f'Chain {c + 1}: Left tail khat exceeds 0.25!')

  if no_warning:
    print('Expectand appears to be sufficiently integrable.\n')
  else:
    print('  Large tail khats suggest that the expectand might')
    print('not be sufficiently integrable.\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.

```
def welford_summary(fs):
  """Welford accumulator for empirical mean and variance of a given sequence"""
  mean = 0
  var = 0

  for n, f in enumerate(fs):
    delta = f - mean
    mean += delta / (n + 1)
    var += delta * (f - mean)

  var /= (len(fs) - 1)

  return [mean, var]
```

```
def check_variances(unpermuted_samples):
  """Check expectand output ensemble for vanishing empirical variance"""
  no_warning = True
  C = len(unpermuted_samples[0,:])
```

```
  for c in range(C):
    fs = unpermuted_samples[:,c]
    var = welford_summary(fs)[1]
    if var < 1e-10:
      no_warning = True
      print(f'Chain {c + 1}: Expectand is constant!\n')

  if no_warning:
    print('Expectand is varying in all Markov chains.')
  else:
    print('  If the expectand is not expected (haha) to be')
    print('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.

```
def split_chain(chain):
  """Split a Markov chain into initial and terminal Markov chains"""
  N = len(chain)
  M = N // 2
  return [ chain[0:M], chain[M:N] ]
```

```
def compute_split_rhat(chains):
  """Compute split hat{R} for an expectand output ensemble across
     a collection of Markov chains"""
```

```
  split_chains = [ c for chain in chains for c in split_chain(chain) ]
  N_chains = len(split_chains)
  N = sum([ len(chain) for chain in split_chains ])

  means = [None] * N_chains
  vars = [None] * N_chains

  for c, chain in enumerate(split_chains):
    summary = welford_summary(chain)
    means[c] = summary[0]
    vars[c] = summary[1]

  total_mean = sum(means) / N_chains
  W = sum(vars) / N_chains
  B = N * sum([ (mean - total_mean)**2 / (N_chains - 1) for mean in means ])

  rhat = math.nan
  if abs(W) > 1e-10:
    rhat = math.sqrt( (N - 1 + B / W) / N )

  return rhat
```

```
def compute_split_rhats(fit, expectand_idxs=None):
  """Compute split hat{R} for all expectand output ensembles across
     a collection of Markov chains"""
  unpermuted_samples = fit.extract(permuted=False)

  input_dims = unpermuted_samples.shape
  N = input_dims[0]
  C = input_dims[1]
  I = input_dims[2]

  if expectand_idxs is None:
    expectand_idxs = range(I)

  bad_idxs = set(expectand_idxs).difference(range(I))
  if len(bad_idxs) > 0:
    print(f'Excluding the invalid expectand indices: {bad_idxs}')
    expectand_idxs = set(expectand_idxs).difference(bad_idxs)

  rhats = []
```

```
  for idx in expectand_idxs:
    chains = [ unpermuted_samples[:,c,idx] for c in range(C) ]
    rhats.append(compute_split_rhat(chains))

  return rhats
```

```
def check_rhat(unpermuted_samples):
  """Check split hat{R} for all expectand output ensembles across
     a collection of Markov chains"""
  C = unpermuted_samples.shape[1]
  chains = [ unpermuted_samples[:,c] for c in range(C) ]
  rhat = compute_split_rhat(chains)

  no_warning = True
  if math.isnan(rhat):
    print('All Markov chains are frozen!')
  elif rhat > 1.1:
    print(f'Split rhat is {rhat:.3f}!')
    no_warning = False

  if no_warning:
    print('Markov chains are consistent with equilibrium.')
  else:
    print('  Split rhat larger than 1.1 is inconsistent with equilibrium.')
```

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.

```
def compute_tauhat(fs):
  """Compute empirical integrated autocorrelation time for a sequence"""
  # Compute empirical autocorrelations
  N = len(fs)
  m, v = welford_summary(fs)
  zs = [ f - m for f in fs ]

  if v < 1e-10:
    return Inf

  B = 2**math.ceil(math.log2(N)) # Next power of 2 after N
  zs_buff = zs + [0] * (B - N)

  Fs = numpy.fft.fft(zs_buff)
  Ss = numpy.abs(Fs)**2
  Rs = numpy.fft.ifft(Ss)

  acov_buff = numpy.real(Rs)
  rhos = acov_buff[0:N] / acov_buff[0]

  # 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[0] + rhos[1]
  for p in range(1, P):
    current_pair_sum = rhos[2 * p] + rhos[2 * p + 1]

    if current_pair_sum < 0:
      rho_sum = sum(rhos[1:(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]     = 0.5 * old_pair_sum
      rhos[2 * p + 1] = 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
```

```
def compute_min_tauhat(fit, expectand_idxs=None):
  """Compute the minimimum empirical integrated autocorrelation time
     across a collection of Markov chains for all expectand output ensembles"""
  unpermuted_samples = fit.extract(permuted=False)

  input_dims = unpermuted_samples.shape
  N = input_dims[0]
  C = input_dims[1]
  I = input_dims[2]

  if expectand_idxs is None:
    expectand_idxs = range(I)

  bad_idxs = set(expectand_idxs).difference(range(I))
  if len(bad_idxs) > 0:
    print(f'Excluding the invalid expectand indices: {bad_idxs}')
    expectand_idxs = set(expectand_idxs).difference(bad_idxs)

  min_int_ac_times = []
  for idx in expectand_idxs:
    int_ac_times = [None] * C
    for c in range(C):
      fs = unpermuted_samples[:,c,idx]
      int_ac_times[c] = compute_tauhat(fs)

    min_int_ac_times.append(min(int_ac_times))
```

```
  return min_int_ac_times
```

```
def check_min_tauhat(unpermuted_samples):
  """Check the empirical integrated autocorrelation times across a
     collection of Markov chains for all expectand output ensembles"""
  input_dims = unpermuted_samples.shape
  N = input_dims[0]
  C = input_dims[1]

  no_warning = True
  for c in range(C):
    fs = unpermuted_samples[:,c]
    int_ac_time = compute_tauhat(fs)
    if (int_ac_time / N) > 0.25:
      print(  f'Chain {c + 1}: The integrated autocorrelation time'
            + 'exceeds 0.25 * N!')
      no_warning = False

  if no_warning:
    print('Autocorrelations within each Markov chain appear to be reasonable.')
  else:
    print('  Autocorrelations in at least one Markov chain are large enough')
    print('that Markov chain Monte Carlo estimates may not be reliable.')
```

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!

```
def check_neff(unpermuted_samples, min_neff_per_chain=100):
  """Check the empirical effective sample size for all expectand output ensembles"""
```

```
  input_dims = unpermuted_samples.shape
  N = input_dims[0]
  C = input_dims[1]

  no_warning = True
  for c in range(C):
    fs = unpermuted_samples[:,c]
    int_ac_time = compute_tauhat(fs)
    neff = N / int_ac_time
    if neff < min_neff_per_chain:
      print(f'Chain {c + 1}: The effective sample size {neff:.1f} is too small!')
      no_warning = False

  if no_warning:
    print('All effective sample sizes are sufficiently large.')
  else:
    print('  If the effective sample size is too small then')
    print('Markov chain Monte Carlo estimators will be imprecise.')
```

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.

```
def check_all_expectand_diagnostics(fit,
                                    expectand_idxs=None,
                                    min_neff_per_chain=100,
                                    exclude_zvar=False):
  """Check all expectand diagnostics"""
  unpermuted_samples = fit.extract(permuted=False)

  input_dims = unpermuted_samples.shape
  N = input_dims[0]
  C = input_dims[1]
  I = input_dims[2]

  expectand_names = fit.flatnames + ["lp__"]

  if expectand_idxs is None:
    expectand_idxs = range(I)
```

```
bad_idxs = set(expectand_idxs).difference(range(I))
if len(bad_idxs) > 0:
  print(f'Excluding the invalid expectand indices: {bad_idxs}')
  expectand_idxs = set(expectand_idxs).difference(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 = expectand_names[idx] + ':\n'

  if exclude_zvar:
    # Check zero variance across all Markov chains for exclusion
    any_zvar = False
    for c in range(C):
      fs = unpermuted_samples[:,c,idx]
      var = welford_summary(fs)[1]
      if var < 1e-10:
        any_zvar = True

    if any_zvar:
      continue

  for c in range(C):
    fs = unpermuted_samples[:,c,idx]

    # Check tail khats in each Markov chain
    khats = compute_tail_khats(fs)
    if khats[0] >= 0.25 and khats[1] >= 0.25:
      no_khat_warning = False
      local_warning = True
      local_message += f'  Chain {c + 1}: Both left and right tail hat{{k}}s' \
                       + f'({khats[0]:.3f}, {khats[1]:.3f}) exceed 0.25!\n'
    elif khats[0] < 0.25 and khats[1] >= 0.25:
      no_khat_warning = False
```

```
      local_warning = True
      local_message +=  f'  Chain {c + 1}: Right tail hat{{k}} ({khats[1]:.3f})' \
                      + ' exceeds 0.25!\n'
    elif khats[0] >= 0.25 and khats[1] < 0.25:
      no_khat_warning = False
      local_warning = True
      local_message +=  f'  Chain {c + 1}: Left tail hat{{k}} ({khats[0]:.3f})' \
                      + ' exceeds 0.25!\n'

    # Check empirical variance in each Markov chain
    var = welford_summary(fs)[1]
    if var < 1e-10:
      no_zvar_warning = False
      local_warning = True
      local_message += f'  Chain {c + 1}: Expectand has vanishing empirical' \
                      + ' variance!\n'

  # Check split Rhat across Markov chains
  chains = [ unpermuted_samples[:,c,idx] for c in range(C) ]
  rhat = compute_split_rhat(chains)

  if math.isnan(rhat):
    local_message += '  Split hat{R} is ill-defined!\n'
  elif rhat > 1.1:
    no_rhat_warning = False
    local_warning = True
    local_message += f'  Split hat{{R}} ({rhat:.3f}) exceeds 1.1!\n'

  for c in range(C):
    # Check empirical integrated autocorrelation time
    fs = unpermuted_samples[:,c,idx]
    int_ac_time = compute_tauhat(fs)
    if (int_ac_time / N) > 0.25:
      no_tauhat_warning = False
      local_warning = True
      local_message +=  f'  Chain {c + 1}: hat{{tau}} per iteration ' \
                      + f'({int_ac_time / N:.3f}) 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 +=  f'  Chain {c + 1}: hat{{ESS}} ({neff:.1f}) is smaller ' \
                        + f'than desired ({min_neff_per_chain:.0f})!\n'

    local_message += '\n'
    if local_warning:
      message += local_message

  if not no_khat_warning:
    message +=  'Large tail hat{k}s suggest that the expectand' \
              + ' might not be sufficiently integrable.\n\n'

  if not no_zvar_warning:
    message +=  'If the expectands are not constant then zero empirical' \
              + ' variance suggests that the Markov' \
              + ' transitions are misbehaving.\n\n'

  if not no_rhat_warning:
    message +=  'Split hat{R} larger than 1.1 is inconsisent with' \
              + ' equilibrium.\n\n'

  if not no_tauhat_warning:
    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 not no_neff_warning:
    message +=  'If hat{ESS} is too small then reliable Markov chain' \
              + ' Monte Carlo estimators may still be too imprecise.\n\n'

  if no_khat_warning and no_zvar_warning and no_rhat_warning \
     and no_tauhat_warning and no_neff_warning:
    message =   'All expectands checked appear to be behaving well enough ' \
              + 'for reliable Markov chain Monte Carlo estimation.\n'

  print(message)
```

```
def expectand_diagnostics_summary(fit,
                                  expectand_idxs=None,
                                  min_neff_per_chain=100,
```

```
                                    exclude_zvar=False):
"""Summarize expectand diagnostics"""
unpermuted_samples = fit.extract(permuted=False)

input_dims = unpermuted_samples.shape
N = input_dims[0]
C = input_dims[1]
I = input_dims[2]

if expectand_idxs is None:
  expectand_idxs = range(I)

bad_idxs = set(expectand_idxs).difference(range(I))
if len(bad_idxs) > 0:
  print(f'Excluding the invalid expectand indices: {bad_idxs}')
  expectand_idxs = set(expectand_idxs).difference(bad_idxs)

failed_idx = []
failed_khat_idx = []
failed_zvar_idx = []
failed_rhat_idx = []
failed_tauhat_idx = []
failed_neff_idx = []

for idx in expectand_idxs:
  if exclude_zvar:
    # Check zero variance across all Markov chains for exclusion
    any_zvar = False
    for c in range(C):
      fs = unpermuted_samples[:,c,idx]
      var = welford_summary(fs)[1]
      if var < 1e-10:
        any_zvar = True
    if any_zvar:
      continue

  for c in range(C):
    # Check tail khats in each Markov chain
    fs = unpermuted_samples[:,c,idx]
    khats = compute_tail_khats(fs)
    if khats[0] >= 0.25 or khats[1] >= 0.25:
```

```
      failed_idx.append(idx)
      failed_khat_idx.append(idx)

    # Check empirical variance in each Markov chain
    var = welford_summary(fs)[1]
    if var < 1e-10:
      failed_idx.append(idx)
      failed_zvar_idx.append(idx)

  # Check split Rhat across Markov chains
  chains = [ unpermuted_samples[:,c,idx] for c in range(C) ]
  rhat = compute_split_rhat(chains)

  if math.isnan(rhat):
    failed_idx.append(idx)
    failed_rhat_idx.append(idx)
  elif rhat > 1.1:
    failed_idx.append(idx)
    failed_rhat_idx.append(idx)

  for c in range(C):
    # Check empirical integrated autocorrelation time
    fs = unpermuted_samples[:,c,idx]
    int_ac_time = compute_tauhat(fs)
    if (int_ac_time / N) > 0.25:
      failed_idx.append(idx)
      failed_tauhat_idx.append(idx)

    # Check empirical effective sample size
    neff = N / int_ac_time
    if neff < min_neff_per_chain:
      failed_idx.append(idx)
      failed_neff_idx.append(idx)

failed_idx = list(numpy.unique(failed_idx))
if len(failed_idx):
  print( f'The expectands {str(failed_idx).replace("[", "").replace("]", "")}' \
        + ' triggered diagnostic warnings.\n')
else:
  print(  'All expectands checked appear to be behaving' \
        + 'well enough for Markov chain Monte Carlo estimation.')
```

```
failed_khat_idx = list(numpy.unique(failed_khat_idx))
if len(failed_khat_idx):
  print( f'The expectands {str(failed_khat_idx).replace("[", "").replace("]", "")}' \
        + ' triggered hat{k} warnings.\n')
  print(  '  Large tail hat{k}s suggest that the expectand' \
        + ' might not be sufficiently integrable.\n\n')

failed_zvar_idx = list(numpy.unique(failed_zvar_idx))
if len(failed_zvar_idx):
  print(  f'The expectands {str(failed_zvar_idx).replace("[", "").replace("]", "")}' \
        + ' triggered zero variance warnings.\n')
  print(  '  If the expectands are not constant then zero empirical' \
        + ' variance suggests that the Markov' \
        + ' transitions are misbehaving.\n\n')

failed_rhat_idx = list(numpy.unique(failed_rhat_idx))
if len(failed_rhat_idx):
  print( f'The expectands {str(failed_rhat_idx).replace("[", "").replace("]", "")}' \
        + ' triggered hat{R} warnings.\n')
  print(  '  Split hat{R} larger than 1.1 is inconsistent with' \
        + ' equilibrium.\n\n')

failed_tauhat_idx = list(numpy.unique(failed_tauhat_idx))
if len(failed_tauhat_idx):
  print( f'The expectands {str(failed_tauhat_idx).replace("[", "").replace("]", "")}' \
        + ' triggered hat{tau} warnings.\n')
  print(  '  hat{tau} larger than a quarter of the Markov chain' \
        + ' length suggests that Markov chain Monte Carlo' \
        + ' estimates may be unreliable.\n\n')

failed_neff_idx = list(numpy.unique(failed_neff_idx))
if len(failed_neff_idx):
  print( f'The expectands {str(failed_neff_idx).replace("[", "").replace("]", "")}' \
        + ' triggered hat{ESS} warnings.\n')
  print(  '  If hat{ESS} is too small then even reliable Markov chain' \
        + ' Monte Carlo estimators may still 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.

```
def compute_rhos(fs):
  """Visualize empirical autocorrelations for a given sequence"""
  # Compute empirical autocorrelations
  N = len(fs)
  m, v = welford_summary(fs)
  zs = [ f - m for f in fs ]

  if v < 1e-10:
    return [1] * N

  B = 2**math.ceil(math.log2(N)) # Next power of 2 after N
  zs_buff = zs + [0] * (B - N)

  Fs = numpy.fft.fft(zs_buff)
  Ss = numpy.abs(Fs)**2
  Rs = numpy.fft.ifft(Ss)

  acov_buff = numpy.real(Rs)
  rhos = acov_buff[0:N] / acov_buff[0]

  # 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 range(1, P):
    current_pair_sum = rhos[2 * p] + rhos[2 * p + 1]

    if current_pair_sum < 0:
```

```
      max_L = 2 * p
      rhos[max_L:N] = [0] * (N - max_L)
      break

    if current_pair_sum > old_pair_sum:
      current_pair_sum = old_pair_sum
      rhos[2 * p]     = 0.5 * old_pair_sum
      rhos[2 * p + 1] = 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

  return rhos
```

```
def plot_empirical_correlogram(ax,
                               unpermuted_fs,
                               max_L,
                               rholim=[-0.2, 1.1],
                               name=""):
  """Plot empirical correlograms for the expectand output ensembels in a
     collection of Markov chains"""
  idxs = [ idx for idx in range(max_L) for r in range(2) ]
  xs = [ idx + delta for idx in range(max_L) for delta in [-0.5, 0.5]]

  colors = [dark, dark_highlight, mid, light_highlight]

  C = (unpermuted_samples.shape)[1]
  for c in range(C):
    fs = unpermuted_fs[:,c]
    rhos = compute_rhos(fs)
    pad_rhos = [ rhos[idx] for idx in idxs ]
    ax.plot(xs, pad_rhos, colors[c % 4], linewidth=2)

  ax.axhline(y=0, linewidth=2, color="#DDDDDD")

  ax.set_title(name)
  ax.set_xlabel("Lag")
  ax.set_xlim(-0.5, max_L + 0.5)
```

```
  ax.set_ylabel("Empirical\nAutocorrelation")
  ax.set_ylim(rholim[0], rholim[1])
  ax.spines["top"].set_visible(False)
  ax.spines["right"].set_visible(False)
```

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.

```
def plot_chain_sep_pairs(unpermuted_f1s, name_x,
                         unpermuted_f2s, name_y):
  """Plot two expectand output ensembles againt each other separated by
     Markov chain """
  input_dims = unpermuted_f1s.shape
  N = input_dims[0]
  C = input_dims[1]

  colors = ["#DCBCBC", "#C79999", "#B97C7C",
            "#A25050", "#8F2727", "#7C0000"]
  cmap = LinearSegmentedColormap.from_list("reds", colors, N=N)

  min_x = min([ min(unpermuted_f1s[:,c]) for c in range(C) ])
  max_x = max([ max(unpermuted_f1s[:,c]) for c in range(C) ])

  min_y = min([ min(unpermuted_f2s[:,c]) for c in range(C) ])
  max_y = max([ max(unpermuted_f2s[:,c]) for c in range(C) ])

  N_plots = C
  N_cols = 2
  N_rows = math.ceil(N_plots / N_cols)
  f, axarr = plot.subplots(N_rows, N_cols)
  k = 0

  for c in range(C):
    idx1 = k // N_cols
    idx2 = k % N_cols
    k += 1
```

```
    axarr[idx1, idx2].scatter(unpermuted_f1s.flatten(),
                              unpermuted_f2s.flatten(),
                              color="#DDDDDD", s=10, zorder=3)
    axarr[idx1, idx2].scatter(unpermuted_f1s[:,c], unpermuted_f2s[:,c],
                              cmap=cmap, c=range(N), s=10, zorder=4)

    axarr[idx1, idx2].set_title(f'Chain {c + 1}')
    axarr[idx1, idx2].set_xlabel(name_x)
    axarr[idx1, idx2].set_xlim([min_x, max_x])
    axarr[idx1, idx2].set_ylabel(name_y)
    axarr[idx1, idx2].set_ylim([min_y, max_y])
    axarr[idx1, idx2].spines["top"].set_visible(False)
    axarr[idx1, idx2].spines["right"].set_visible(False)

  plot.subplots_adjust(hspace=1.0, wspace=0.5)
  plot.show()
```

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.

```
def pushforward_chains(chains, expectand):
  """Evaluate an expectand along a Markov chain"""
  return [ [ expectand(x) for x in chain ] for chain in chains ]
```

```
def mcmc_est(fs):
  """Estimate expectand expectation value from a Markov chain"""
  N = len(fs)
  if N == 1:
    return [fs[0], 0, math.nan]

  summary = welford_summary(fs)

  if summary[1] == 0:
    return [summary[0], 0, math.nan]

  int_ac_time = compute_tauhat(fs)
```

```
  neff = N / int_ac_time
  return [summary[0], math.sqrt(summary[1] / neff), neff]
```

```
def ensemble_mcmc_est(chains):
  """Estimate expectand exectation value from a collection of Markov chains"""
  C = len(chains)
  chain_ests = [ mcmc_est(chain) for chain in chains ]

  # Total effective sample size
  total_ess = sum([ est[2] for est in chain_ests ])

  if math.isnan(total_ess):
    m  = numpy.mean([ est[0] for est in chain_ests ])
    se = numpy.mean([ est[1] for est in chain_ests ])
    return [m, se, math.nan]

  # Ensemble average weighted by effective sample size
  mean = sum([ est[0] * est[2] for est in chain_ests ]) / 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 = [0] * C

  for c in range(C):
    est = chain_ests[c]
    chain_var = est[2] * est[1]**2
    var_update = (est[0] - mean)**2
    vars[c] = est[2] * (var_update + chain_var)
  var = sum(vars) / total_ess

  return [mean, math.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.

```
def plot_pushforward_hist(ax, unpermuted_samples, B, flim=None, name="f"):
  """Plot pushforward histogram of a given expectand using Markov chain
     Monte Carlo estimators to estimate the output bin probabilities"""
```

```
if flim is None:
  # Automatically adjust histogram binning to range of outputs
  min_f = min(unpermuted_samples.flatten())
  max_f = max(unpermuted_samples.flatten())

  # Add bounding bins
  delta = (max_f - min_f) / B
  min_f = min_f - delta
  max_f = max_f + delta
  flim = [min_f, max_f]

  bins = numpy.arange(min_f, max_f + delta, delta)
  B = B + 2
else:
  delta = (flim[1] - flim[0]) / B
  bins = numpy.arange(flim[0], flim[1] + delta, delta)

mean_p = [0] * B
delta_p = [0] * B

C = (unpermuted_samples.shape)[1]
chains = [ unpermuted_samples[:,c] for c in range(C) ]

for b in range(B):
  def bin_indicator(x):
    return 1.0 if bins[b] <= x and x < bins[b + 1] else 0.0

  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[0] / width
  delta_p[b] = est[1] / width

idxs = [ idx for idx in range(B) for r in range(2) ]
xs = [ bins[b + o] for b in range(B) for o in range(2) ]

lower_inter = [ max(mean_p[idx] - 2 * delta_p[idx], 0)         for idx in idxs ]
upper_inter = [ min(mean_p[idx] + 2 * delta_p[idx], 1 / width) for idx in idxs ]
```

```
  min_y =        min(lower_inter)
  max_y = 1.05 * max(upper_inter)

  ax.fill_between(xs, lower_inter, upper_inter,
                  facecolor=light, color=light)
  ax.plot(xs, [ mean_p[idx] for idx in idxs ], color=dark, linewidth=2)

  ax.set_xlim(flim)
  ax.set_xlabel(name)
  ax.set_ylim([min_y, max_y])
  ax.set_ylabel("Estimated Bin\nProbabilities")
  ax.get_yaxis().set_visible(False)
  ax.spines["top"].set_visible(False)
  ax.spines["left"].set_visible(False)
  ax.spines["right"].set_visible(False)
```

4 Demonstration

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

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

```
with open('stan_programs/simu_logistic_reg.stan', 'r') as file:
  lines = file.readlines()
  for line in lines:
    print(line),
```

```
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')');
  }
}
```

```
model = compile_model('stan_programs/simu_logistic_reg.stan')
simu = model.sampling(iter=1, warmup=0, chains=1, chain_id=[1],
                      refresh=1000, seed=4838282, algorithm="Fixed_param")

X = simu.extract()['X'][0]
y = simu.extract()['y'][0].astype(numpy.int64)

data = dict(M = 3, N = 1000, x0 = [-1, 0, 1], X = X, y = y)
```

```
Using cached StanModel
Iteration: 1 / 1 [100%]  (Sampling)

 Elapsed Time: 0 seconds (Warm-up)
               0.00166 seconds (Sampling)
               0.00166 seconds (Total)
```

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.

```
with open('stan_programs/bernoulli_linear.stan', 'r') as file:
  lines = file.readlines()
  for line in lines:
    print(line),
```

```
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.

```
import scipy.stats as stats
numpy.random.seed(seed=48383499)

interval_inits = [None] * 4

for c in range(4):
  beta = [0, 0, 0]
  alpha = stats.norm.rvs(0.5, 0.1, size=1)[0]
  interval_inits[c] = dict(alpha = alpha, beta = beta)
```

```
model = compile_model('stan_programs/bernoulli_linear.stan')
fit = model.sampling(data=data, seed=8438338, warmup=1000, iter=2024,
                     chain_id=[1, 2, 3, 4], refresh=0, init=interval_inits)
```

```
Using cached StanModel
```

```
Gradient evaluation took 0.000198 seconds
1000 transitions using 10 leapfrog steps per transition would take 1.98 seconds.
Adjust your expectations accordingly!


Gradient evaluation took 0.000194 seconds
1000 transitions using 10 leapfrog steps per transition would take 1.94 seconds.
Adjust your expectations accordingly!


Gradient evaluation took 0.000171 seconds
1000 transitions using 10 leapfrog steps per transition would take 1.71 seconds.
Adjust your expectations accordingly!


Gradient evaluation took 0.000165 seconds
1000 transitions using 10 leapfrog steps per transition would take 1.65 seconds.
Adjust your expectations accordingly!


 Elapsed Time: 0.60704 seconds (Warm-up)
               0.478339 seconds (Sampling)
               1.08538 seconds (Total)


 Elapsed Time: 0.689018 seconds (Warm-up)
               0.495555 seconds (Sampling)
               1.18457 seconds (Total)

 Elapsed Time: 0.743199 seconds (Warm-up)
               0.544426 seconds (Sampling)
               1.28763 seconds (Total)

 Elapsed Time: 0.721348 seconds (Warm-up)
               3.28321 seconds (Sampling)
               4.00456 seconds (Total)
```

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.

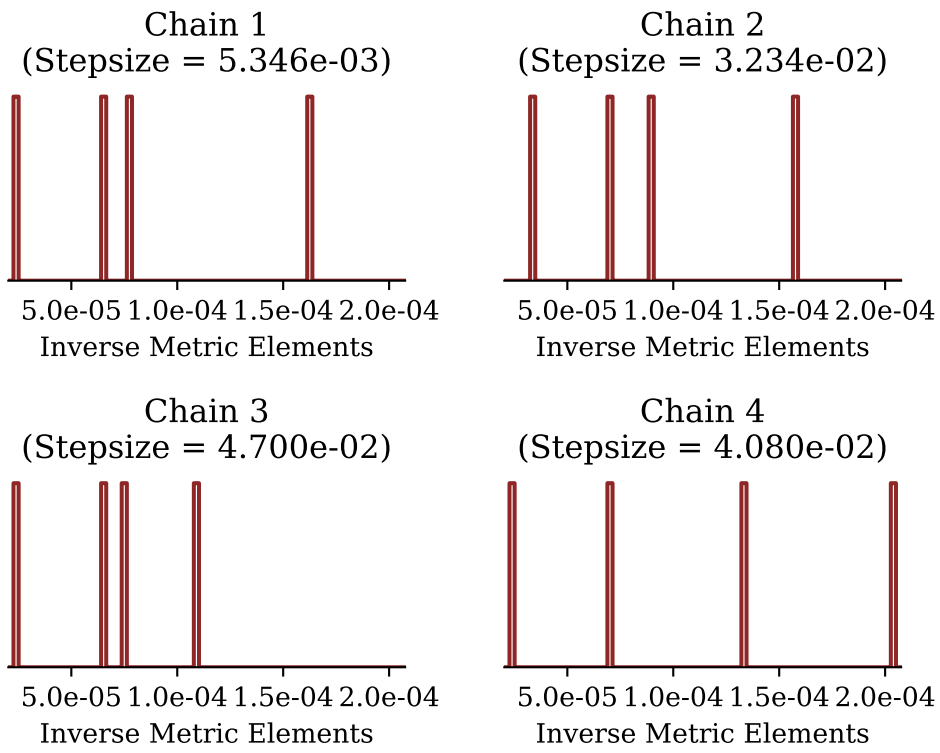
```
check_all_hmc_diagnostics(fit)
```

```
4056 of 4096 iterations ended with a divergence (99.02%)
  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 3: Average proxy acceptance statistic (0.716)
         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 quite 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 step size in the third Markov chain is slightly larger than the others which explains the lower average proxy acceptance statistic. We can also see that the first Markov chain has a much smaller step size than the other which results in an overly conservative average proxy acceptance statistic.

```
display_stepsizes(fit)
```

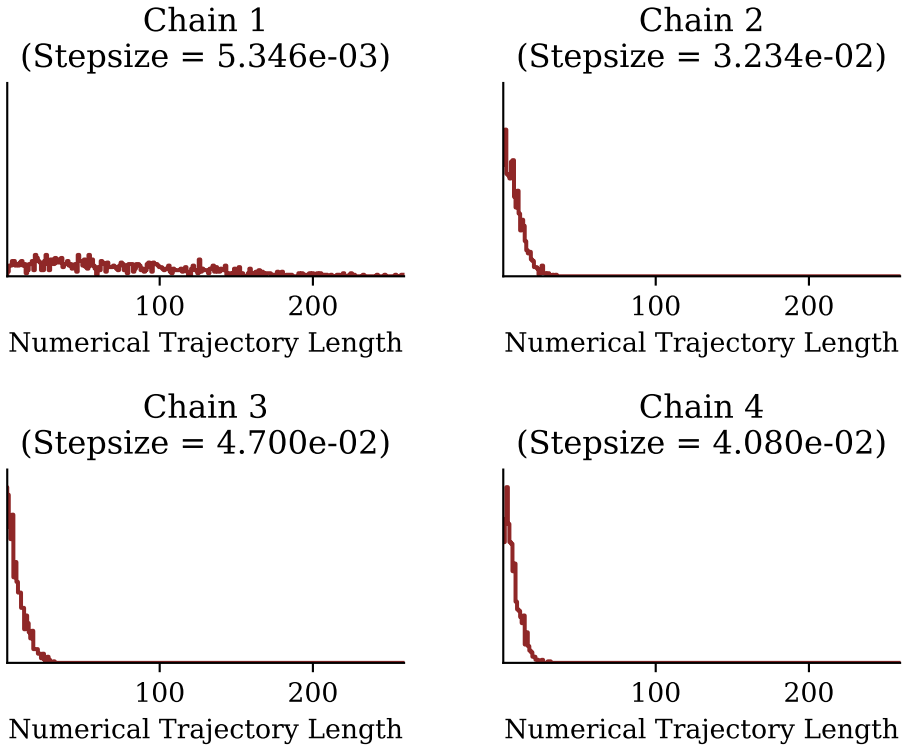
```
Chain 1: Integrator Step Size = 5.35e-03
Chain 2: Integrator Step Size = 3.23e-02
Chain 3: Integrator Step Size = 4.70e-02
Chain 4: Integrator Step Size = 4.08e-02
```

```
display_ave_accept_proxy(fit)
```

```
Chain 1: Average proxy acceptance statistic = 0.968
Chain 2: Average proxy acceptance statistic = 0.760
Chain 3: Average proxy acceptance statistic = 0.716
Chain 4: Average proxy acceptance statistic = 0.737
```

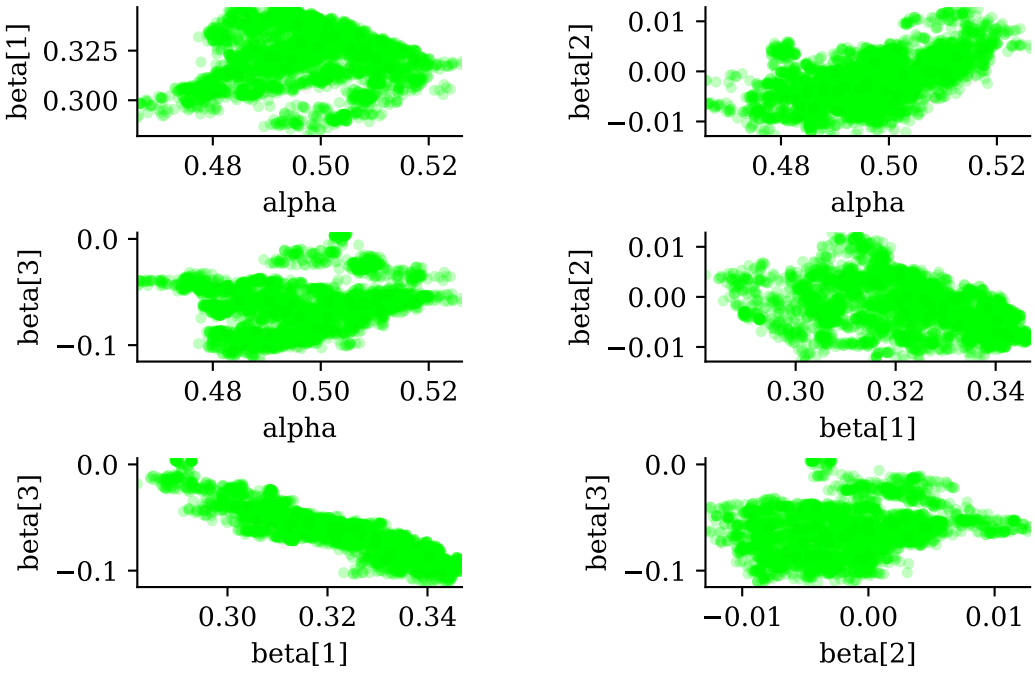
That one Markov chain with the smaller adapted step size requires much longer, and more expensive, numerical trajectories, than than the other three Markov chains in order to attain the same exploration.

```
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,
               ["alpha", "beta[1]", "beta[2]", "beta[3]"],
               [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 let's start by looking at the expectand diagnostics summary instead of the full details.

```
expectand_diagnostics_summary(fit)
```

```
The expectands 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,

The expectands 803 triggered hat{k} warnings.

  Large tail hat{k}s suggest that the expectand might not be sufficiently integrable.


The expectands 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,

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

```
The expectands 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,

  If hat{ESS} is too small then even reliable Markov chain Monte Carlo estimators may still b
```

That is a lot of diagnostic failures. To avoid overwhelming ourselves with too many detailed diagnosic messages let's focus on the four parameter expectands.

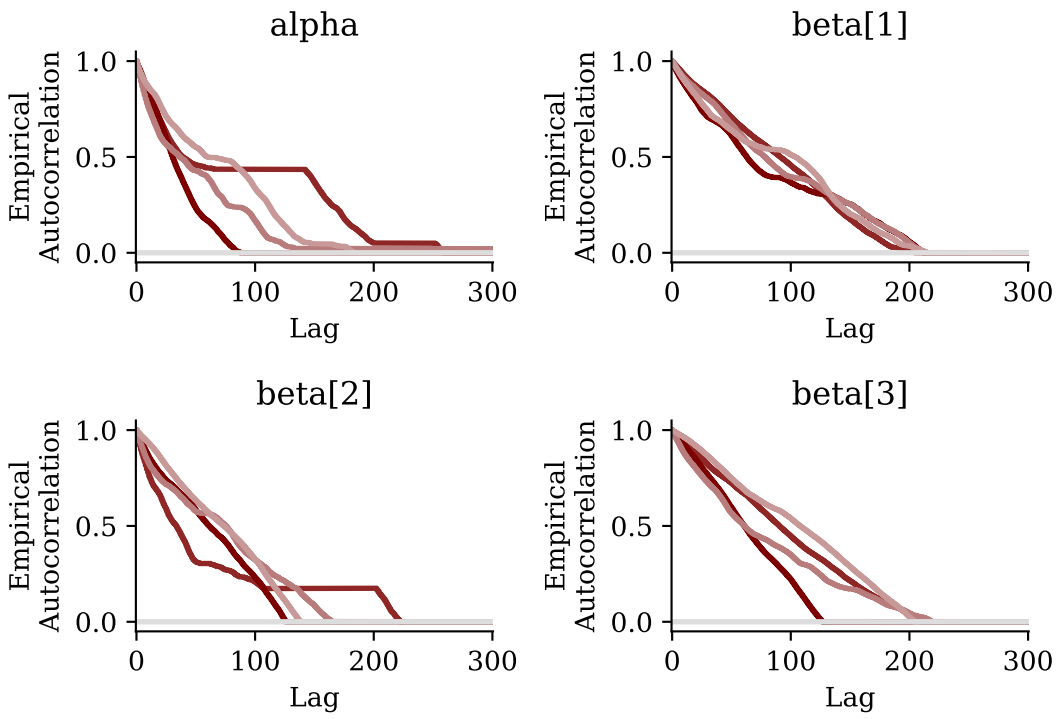
```
check_all_expectand_diagnostics(fit, range(4))
```

```
alpha:
  Split hat{R} (1.600) exceeds 1.1!
  Chain 1: hat{ESS} (5.8) is smaller than desired (100)!
  Chain 2: hat{ESS} (14.9) is smaller than desired (100)!
  Chain 3: hat{ESS} (9.9) is smaller than desired (100)!
  Chain 4: hat{ESS} (7.4) is smaller than desired (100)!

beta[1]:
  Split hat{R} (1.646) exceeds 1.1!
  Chain 1: hat{ESS} (5.6) is smaller than desired (100)!
  Chain 2: hat{ESS} (6.0) is smaller than desired (100)!
  Chain 3: hat{ESS} (5.6) is smaller than desired (100)!
  Chain 4: hat{ESS} (5.6) is smaller than desired (100)!

beta[2]:
  Split hat{R} (1.163) exceeds 1.1!
  Chain 1: hat{ESS} (8.1) is smaller than desired (100)!
  Chain 2: hat{ESS} (8.3) is smaller than desired (100)!
  Chain 3: hat{ESS} (7.1) is smaller than desired (100)!
  Chain 4: hat{ESS} (7.2) is smaller than desired (100)!

beta[3]:
  Split hat{R} (1.731) exceeds 1.1!
  Chain 1: hat{ESS} (5.4) is smaller than desired (100)!
  Chain 2: hat{ESS} (8.1) is smaller than desired (100)!
  Chain 3: hat{ESS} (6.5) is smaller than desired (100)!
  Chain 4: hat{ESS} (4.9) is smaller than desired (100)!

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

If hat{ESS} is too small then reliable Markov chain Monte Carlo estimators may still be too i
```

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.

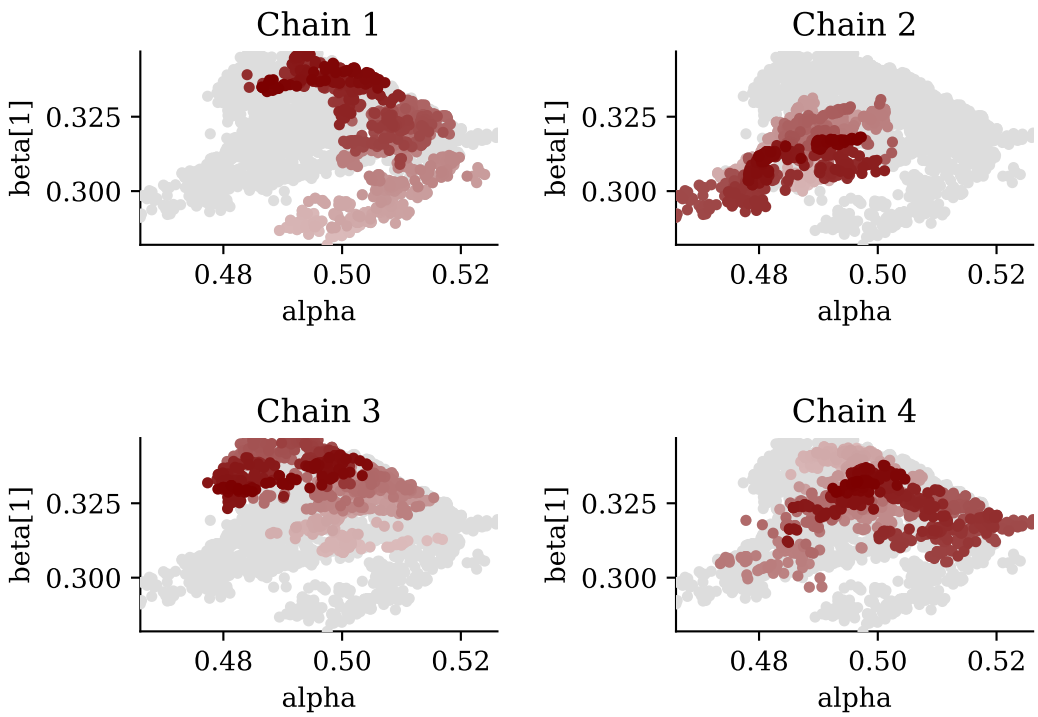
```
unpermuted_samples = fit.extract(permuted=False)

f, axarr = plot.subplots(2, 2)

plot_empirical_correlogram(axarr[0, 0], unpermuted_samples[:,:,0],
                           300, [-0.05, 1.05], "alpha")
plot_empirical_correlogram(axarr[0, 1], unpermuted_samples[:,:,1],
                           300, [-0.05, 1.05], "beta[1]")
plot_empirical_correlogram(axarr[1, 0], unpermuted_samples[:,:,2],
                           300, [-0.05, 1.05], "beta[2]")
plot_empirical_correlogram(axarr[1, 1], unpermuted_samples[:,:,3],
                           300, [-0.05, 1.05], "beta[3]")

plot.subplots_adjust(wspace=0.5, hspace=0.75)
plot.show()
```



Regardless of whether or not these Markov chains are stationary they are extremely autocorrelated. Even assuming stationarity we wouldn'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.

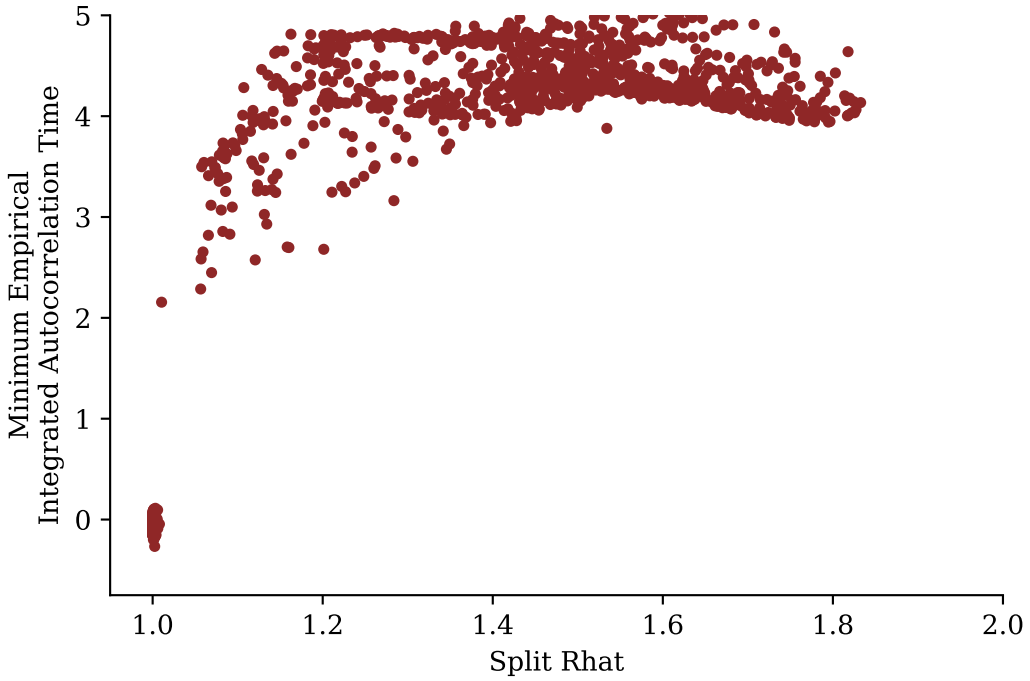
This is consistent with the constraint violations breaking the coherent, gradient-driven exploration of Hamiltonian Monte Carlo so that the Markov chains devolve into diffuse random walks. Indeed looking at the chain-separated pairs plots we see the spatial color continuity characteristic of a random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

```
plot_chain_sep_pairs(unpermuted_samples[:,:,0], "alpha",
                     unpermuted_samples[:,:,1], "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.

```
rhats = compute_split_rhats(fit)
min_tauhats = compute_min_tauhat(fit)

plot.scatter(rhats, [ math.log(y) for y in min_tauhats ], color=dark, s=10)
plot.gca().set_xlim([0.95, 2])
plot.gca().set_xlabel("Split Rhat")
plot.gca().set_ylim([-0.75, 5])
plot.gca().set_ylabel("Minimum Empirical\nIntegrated Autocorrelation Time")
plot.gca().spines["top"].set_visible(False)
plot.gca().spines["right"].set_visible(False)

plot.show()
```

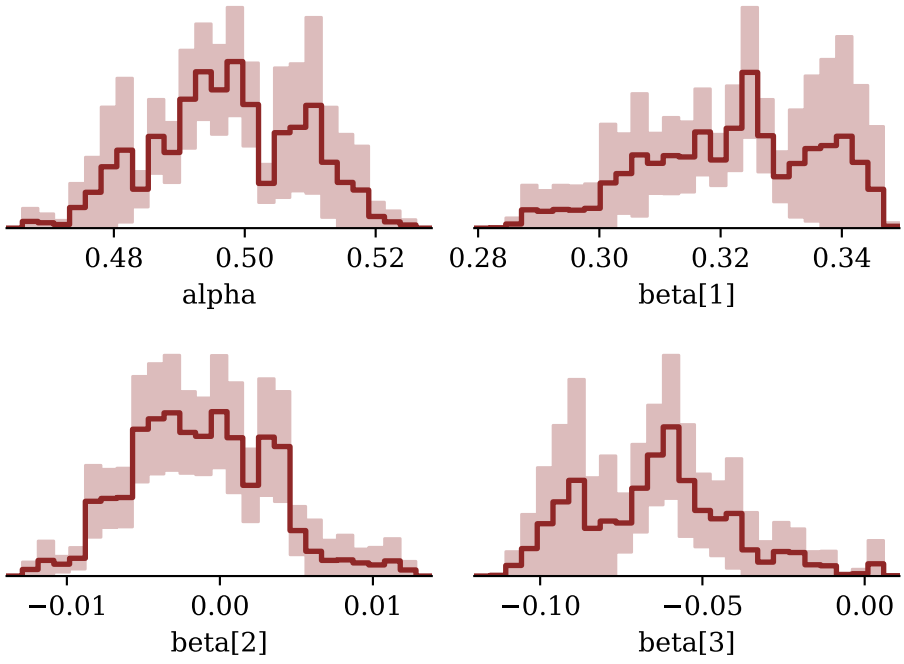


Every expectand with a large split $\hat{R}$s also exhibits a large value the minimum empirical integrated autocorrelation time, confirming that the latter are due to our Markov chains not containing enough information.

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.

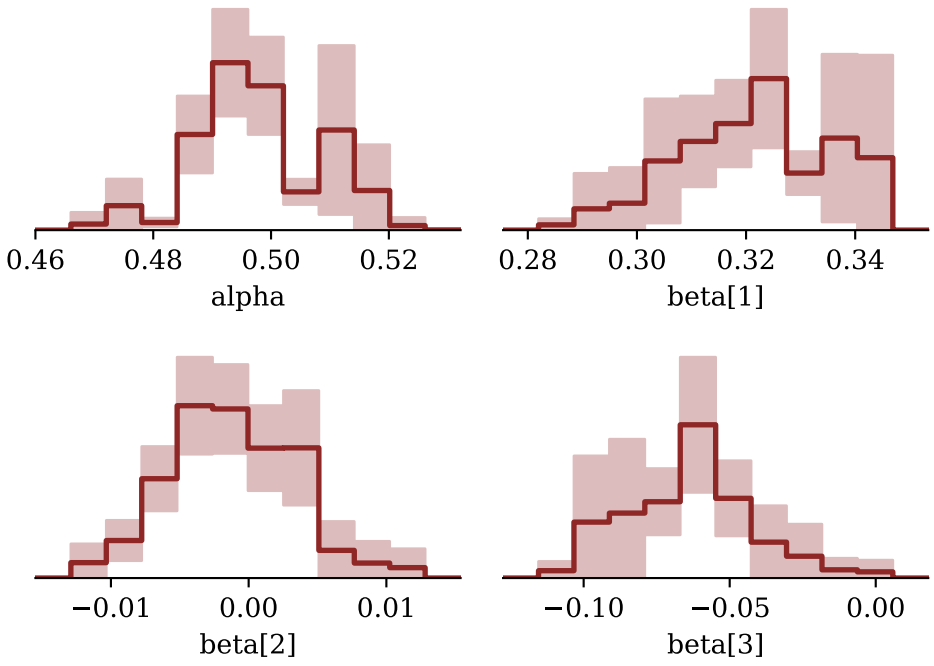
```
f, axarr = plot.subplots(2, 2)

plot_pushforward_hist(axarr[0, 0], unpermuted_samples[:,:,0], 25, name="alpha")
plot_pushforward_hist(axarr[0, 1], unpermuted_samples[:,:,1], 25, name="beta[1]")
plot_pushforward_hist(axarr[1, 0], unpermuted_samples[:,:,2], 25, name="beta[2]")
plot_pushforward_hist(axarr[1, 1], unpermuted_samples[:,:,3], 25, name="beta[3]")

plot.subplots_adjust(wspace=0.1, hspace=0.5)
plot.show()
```



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.

```
f, axarr = plot.subplots(2, 2)

plot_pushforward_hist(axarr[0, 0], unpermuted_samples[:,:,0], 10, name="alpha")
plot_pushforward_hist(axarr[0, 1], unpermuted_samples[:,:,1], 10, name="beta[1]")
plot_pushforward_hist(axarr[1, 0], unpermuted_samples[:,:,2], 10, name="beta[2]")
plot_pushforward_hist(axarr[1, 1], unpermuted_samples[:,:,3], 10, name="beta[3]")

plot.subplots_adjust(wspace=0.1, hspace=0.5)
plot.show()
```



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

```
from watermark import watermark
print(watermark())
```

```
Last updated: 2022-11-27T11:23:29.807115-05:00

Python implementation: CPython
Python version       : 3.9.6
IPython version      : 8.3.0

Compiler    : Clang 12.0.0 (clang-1200.0.32.29)
OS          : Darwin
Release     : 19.6.0
Machine     : x86_64
Processor   : i386
CPU cores   : 16
Architecture: 64bit
```

```
print(watermark(packages="matplotlib,numpy,pystan"))
```

```
matplotlib: 3.5.2
numpy     : 1.22.3
pystan    : 2.19.1.1
```