Markov Chain Monte Carlo Diagnostics

Michael Betancourt

2023-01-01

Table of contents

1 Extraction **2**

2 Hamiltonian Monte Carlo Diagnostics **3**
2.1 Check Hamiltonian Monte Carlo Diagnostics 4
2.2 Integrator Inverse Metric Elements 8
2.3 Integrator Step Sizes 10
2.4 Numerical Trajectory Lengths 10
2.5 Average Proxy Acceptance Statistic 13
2.6 Divergence-Labeled Pairs Plot 13

3 Expectand Diagnostic Functions **20**
3.1 xihat 20
3.2 Frozen Chains 24
3.3 Split Rhat 26
3.4 Integrated Autocorrelation Time 28
3.5 All Expectand Diagnostics 34
3.6 Empirical Autocorrelation Visualization 43
3.7 Chain-Separated Pairs Plot 46

4 Markov Chain Monte Carlo Estimation **48**

5 Demonstration **57**

License **86**

Original Computing Environment **86**

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.

1 Extraction

Starting in version 3 `PyStan` stores all outputs, including the Markov chain Monte Carlo samples and diagnostics, together in a single array that can be accessed from the `_draws` member of a `StanFit` object. The variable names are stored separately in various `*_param_names` member variables. The raw output from each Markov chain is also saved in a binary format that can be accessed with the `stan_outputs` member variable.

To facilitate the analysis of Stan output I've included my own custom extract functions that format the sample and diagnostic outputs into dictionaries, with one key for each expectand or Hamiltonian Monte Carlo diagnostic. The elements themselves are two-dimensional arrays with the first index denoting the individual Markov chains and the second index denoting the iterations within an individual Markov chain.

```
# Extract unpermuted expectand values from a StanFit object and format
# them for convenient access
# @param stan_fit A StanFit object
# @return A dictionary of two-dimensional arrays for each expectand in
#         the StanFit object.  The first dimension of each element
#         indexes the Markov chains and the second dimension indexes the
#         sequential states within each Markov chain.
def extract_expectand_vals(stan_fit):
  nom_params = stan_fit._draws
  offset = len(stan_fit.sample_and_sampler_param_names)

  base_names = stan_fit.constrained_param_names
  formatted_names = []
  for base_name in base_names:
    name = re.sub('\.', '[', base_name, count=1)
    name = re.sub('\.', ',', name)
    if '[' in name:
      name += ']'
    formatted_names.append(name)
```

```
  params = { name: numpy.transpose(nom_params[k + offset,:,:])
             for k, name in enumerate(formatted_names) }
  return params
```

```
# Extract Hamiltonian Monte Carlo diagnostics values from a StanFit
# object and format them for convenient access
# @param stan_fit A StanFit object
# @return A dictionary of two-dimensional arrays for each expectand in
#         the StanFit object.  The first dimension of each element
#         indexes the Markov chains and the second dimension indexes the
#         sequential states within each Markov chain.
def extract_hmc_diagnostics(stan_fit):
  d_names = ['divergent__', 'treedepth__', 'n_leapfrog__',
             'stepsize__', 'energy__', 'accept_stat__' ]
  for dn in d_names:
    if dn not in stan_fit.sample_and_sampler_param_names:
      print(f'Diagnostic variable {dn} not found in stan_fit.')
      return

  d_idxs = [ idx for dn in d_names
             for idx, sn
             in enumerate(stan_fit.sample_and_sampler_param_names)
             if sn == dn ]

  params = { name: numpy.transpose(stan_fit._draws[idx,:,:])
           for idx, name in zip(d_idxs, d_names) }
  return params
```

If users are able to modify these functions to accept the output from other interfaces to Stan and return the same output format then all of the following functions will be immediately available. That is all except for the `plot_inv_metric` function which does require a separate function for extracting adaptation information.

2 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.

2.1 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 `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.

```
# Check all Hamiltonian Monte Carlo Diagnostics
# for an ensemble of Markov chains
# @param diagnostics A dictionary of two-dimensional arrays for
#                    each expectand.  The first dimension of each
#                    element indexes the Markov chains and the
#                    second dimension indexes the sequential
#                    states within each Markov chain.
# @param adapt_target Target acceptance proxy statistic for step size
#                     adaptation.
# @param max_treedepth The maximum numerical trajectory treedepth
# @param max_width Maximum line width for printing
def check_all_hmc_diagnostics(diagnostics,
                              adapt_target=0.801,
                              max_treedepth=10,
                              max_width=72):
  """Check all Hamiltonian Monte Carlo diagnostics for an
     ensemble of Markov chains"""
  validate_dict_of_arrays(diagnostics, 'diagnostics')

  no_warning = True
  no_divergence_warning = True
  no_treedepth_warning = True
  no_efmi_warning = True
  no_accept_warning = True

  messages = []

  C = diagnostics['divergent__'].shape[0]
  S = diagnostics['divergent__'].shape[1]

  for c in range(C):
    local_messages = []

    # Check for divergences
    n_div = sum(diagnostics['divergent__'][c])

    if n_div > 0:
      no_warning = False
      no_divergence_warning = False
      local_messages.append(f'  Chain {c + 1}: {n_div:.0f} of {S} '
```

```
                          f'transitions ({n_div / S:.2%}) diverged.')

  # Check for tree depth saturation
  n_tds = sum([ td >= max_treedepth
                for td in diagnostics['treedepth__'][c] ])

  if n_tds > 0:
    no_warning = False
    no_treedepth_warning = False
    local_messages.append(f'  Chain {c + 1}: {n_tds:.0f} of {S} '
                          f'transitions ({n_tds / S:.2%}) saturated '
                          f'the maximum treedepth of {max_treedepth}.')

  # Check the energy fraction of missing information (E-FMI)
  energies = diagnostics['energy__'][c]
  numer = sum( [ (energies[i] - energies[i - 1])**2
                 for i in range(1, len(energies)) ] ) / S
  denom = numpy.var(energies)
  if numer / denom < 0.2:
    no_warning = False
    no_efmi_warning = False
    local_messages.append(f'  Chain {c + 1}: '
                          f'E-FMI = {numer / denom:.3f}.')

  # Check convergence of the stepsize adaptation
  ave_accept_proxy = numpy.mean(diagnostics['accept_stat__'][c])
  if ave_accept_proxy < 0.9 * adapt_target:
    no_warning = False
    no_accept_warning = False
    local_message = (f'  Chain {c + 1}: Average proxy acceptance '
                     f'statistic ({ave_accept_proxy:.3f}) is smaller '
                     f'than 90% of the target ({adapt_target:.3f}).')
    local_message = textwrap.wrap(local_message, max_width)
    local_messages += local_message

  if len(local_messages) > 0:
    messages.append(local_messages)
    messages.append([' '])

if no_warning:
  desc = ('All Hamiltonian Monte Carlo diagnostics are consistent '
          'with accurate Markov chain Monte Carlo.')
```

```
  desc = textwrap.wrap(desc, max_width)
  messages.append(desc)
  messages.append([' '])

if not no_divergence_warning:
  desc = ('Divergent Hamiltonian transitions result from '
          'unstable numerical trajectories.  These '
          'instabilities are often due to degenerate target '
          'geometry, especially "pinches".  If there are '
          'only a small number of divergences then running '
          'with adept_delta larger '
          f'than {adapt_target:.3f} may reduce the '
          'instabilities at the cost of more expensive '
          'Hamiltonian transitions.')
  desc = textwrap.wrap(desc, max_width)
  messages.append(desc)
  messages.append([' '])

if not no_treedepth_warning:
  desc = ('Numerical trajectories that saturate the '
          'maximum treedepth have terminated prematurely.  '
          f'Increasing max_depth above {max_treedepth} '
          'should result in more expensive, but more '
          'efficient, Hamiltonian transitions.')
  desc = textwrap.wrap(desc, max_width)
  messages.append(desc)
  messages.append([' '])

if not no_efmi_warning:
  desc = ('E-FMI below 0.2 arise when a funnel-like geometry '
          'obstructs how effectively Hamiltonian trajectories '
          'can explore the target distribution.')
  desc = textwrap.wrap(desc, max_width)
  messages.append(desc)
  messages.append([' '])

if not no_accept_warning:
  desc = ('A small average proxy acceptance statistic '
          'indicates that the adaptation of the numerical '
          'integrator step size failed to converge.  This is '
          'often due to discontinuous or imprecise '
          'gradients.')
```

```
    desc = textwrap.wrap(desc, max_width)
    messages.append(desc)
    messages.append([' '])

  print('\n'.join([ '\n'.join(m) for m in messages ]))
```

2.2 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 outcome of inverse metric adaptation
# @param stan_fit A StanFit object
# @params B The number of bins for the inverse metric element histograms.
def plot_inv_metric(stan_fit, B=25):
  """Plot outcome of inverse metric adaptation"""

  C = len(stan_fit.stan_outputs)
  stepsize_header = b'["Adaptation terminated"]}\n'
  inv_metric_header = b'["Diagonal elements of inverse mass matrix:"]}\n'

  stepsizes = []
  inv_metric_elems = []

  for c in range(C):
    first_split = stan_fit.stan_outputs[c].partition(stepsize_header)[2]
    stepsize_info = first_split.partition(b'\n')[0]
    stepsizes.append(float(eval(stepsize_info)['values'][0].split(' = ')[1]))

    first_split = stan_fit.stan_outputs[c].partition(inv_metric_header)[2]
    inv_metric_str = first_split.partition(b'\n')[0]
    inv_metric_dict = eval(inv_metric_str)['values'][0].split(', ')
    inv_metric_elems.append([ float(x) for x in inv_metric_dict ])

  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, layout="constrained")
k = 0

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]

  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'
                              f' = {stepsizes[c]:.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.show()
```

Note that the adaptation information may be accessed differently in other Stan interfaces, in which case this function would have to be modified accordingly.

2.3 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.

```
# Display adapted symplectic integrator step sizes
# @param diagnostics A dictionary of two-dimensional arrays for
#                    each expectand.  The first dimension of each
#                    element indexes the Markov chains and the
#                    second dimension indexes the sequential
#                    states within each Markov chain.
def display_stepsizes(diagnostics):
  """Display adapted symplectic integrator step sizes"""
  validate_dict_of_arrays(diagnostics, 'diagnostics')

  stepsizes = diagnostics['stepsize__']
  C = stepsizes.shape[0]

  for c in range(C):
    stepsize = stepsizes[c, 1]
    print(f'Chain {c + 1}: Integrator Step Size = {stepsize:.2e}')
```

2.4 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.

```
# Display symplectic integrator trajectory lengths
# @ax Matplotlib axis object
# @param diagnostics A dictionary of two-dimensional arrays for
#                    each expectand.  The first dimension of each
#                    element indexes the Markov chains and the
```

```
#                    second dimension indexes the sequential
#                    states within each Markov chain.
# @param nlim Optional histogram range
def plot_num_leapfrogs(ax, diagnostics, nlim=None):
  """Display symplectic integrator trajectory lengths"""
  validate_dict_of_arrays(diagnostics, 'diagnostics')

  lengths = diagnostics['n_leapfrog__']

  C = lengths.shape[0]
  colors = [dark_highlight, dark, mid_highlight, mid, light_highlight]

  vals_counts = [ numpy.unique(lengths[c], return_counts=True)
                  for c in range(C) ]
  max_n = max([ max(a[0]) for a in vals_counts ]).astype(numpy.int64) + 1
  max_counts = max([ max(a[1]) for a in vals_counts ])

  if nlim is None:
    nlim = [0.5, max_n + 0.5]

  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]]

  for c in range(C):
    counts = numpy.histogram(lengths[c],
                             bins=numpy.arange(0.5, max_n + 1.5, 1))[0]
    ys = counts[idxs]

    ax.plot(xs, ys, colors[c])

  ax.set_xlabel("Numerical Trajectory Lengths")
  ax.set_xlim(nlim)
  ax.set_ylabel("")
  ax.get_yaxis().set_visible(False)
  ax.set_ylim([0, 1.1 * max_counts])
  ax.spines["top"].set_visible(False)
  ax.spines["right"].set_visible(False)
```

```
# Display symplectic integrator trajectory lengths by Markov chain
# @param diagnostics A dictionary of two-dimensional arrays for
#                    each expectand.  The first dimension of each
#                    element indexes the Markov chains and the
```

```
#                    second dimension indexes the sequential
#                    states within each Markov chain.
def plot_num_leapfrogs_by_chain(diagnostics):
  """Display symplectic integrator trajectory lengths"""
  validate_dict_of_arrays(diagnostics, 'diagnostics')

  lengths = diagnostics['n_leapfrog__']
  C = lengths.shape[0]

  vals_counts = [ numpy.unique(lengths[c], return_counts=True)
                  for c in range(C) ]
  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]]

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

  for c in range(C):
    counts = numpy.histogram(lengths[c],
                             bins=numpy.arange(0.5, max_n + 1.5, 1))[0]
    ys = counts[idxs]

    eps = diagnostics['stepsize__'][c][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 Lengths")
    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.show()
```

2.5 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 empirical average of the proxy acceptance statistic across
# each Markov chain
# @param diagnostics A dictionary of two-dimensional arrays for
#                    each expectand.  The first dimension of each
#                    element indexes the Markov chains and the
#                    second dimension indexes the sequential
#                    states within each Markov chain.
def display_ave_accept_proxy(diagnostics):
  """Display empirical average of the proxy acceptance statistic
     across each Markov chain"""
  validate_dict_of_arrays(diagnostics, 'diagnostics')

  proxy_stats = diagnostics['accept_stat__']
  C = proxy_stats.shape[0]

  for c in range(C):
    proxy_stat = numpy.mean(proxy_stats[c,:])
    print(  f'Chain {c + 1}: Average proxy acceptance '
          + f'statistic = {proxy_stat:.3f}')
```

2.6 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.

```
# Apply transformation identity, log, or logit transformation to
# named values and flatten the output.  Transformation defaults to
# identity if name is not included in `transforms` dictionary.  A
# ValueError is thrown if values are not properly constrained.
# @param name Expectand name.
# @param expectand_vals_dict A dictionary of two-dimensional arrays for
#                            each expectand.  The first dimension of
#                            each element indexes the Markov chains and
#                            the second dimension indexes the sequential
#                            states within each Markov chain.
# @param transforms A dictionary with expectand names for keys and
#                   transformation flags for values.
# @return The transformed expectand name and a one-dimensional array of
#         flattened transformation outputs.
def apply_transform(name, expectand_vals_dict, transforms):
  t = transforms.get(name, 0)
  transformed_name = ""
  transformed_vals = 0
  if t == 0:
    transformed_name = name
    transformed_vals = expectand_vals_dict[name].flatten()
  elif t == 1:
    if numpy.amin(expectand_vals_dict[name]) <= 0:
      raise ValueError( 'Log transform requested for expectand '
                        f'{name} but expectand values are not strictly '
                         'positive.')
    transformed_name = f'log({name})'
    transformed_vals = [ math.log(x) for x in
                         expectand_vals_dict[name].flatten() ]
  elif t == 2:
    if (   numpy.amin(expectand_vals_dict[name]) <= 0
        or numpy.amax(expectand_vals_dict[name]) >= 1):
      raise ValueError( 'Logit transform requested for expectand '
                        f'{name} but expectand values are not strictly '
                         'confined to the unit interval.')
    transformed_name = f'logit({name})'
    transformed_vals = [ math.log(x / (1 - x)) for x in
                         expectand_vals_dict[name].flatten() ]
  return transformed_name, transformed_vals
```

```
# Plot pairwise scatter plots with non-divergent and divergent
# transitions separated by color
```

```
# @param x_names A list of expectand names to be plotted on the x axis.
# @param y_names A list of expectand names to be plotted on the y axis.
# @param expectand_vals_dict A dictionary of two-dimensional arrays for
#                            each expectand.  The first dimension of
#                            each element indexes the Markov chains and
#                            the second dimension indexes the sequential
#                            states within each Markov chain.
# @param diagnostics A dictionary of two-dimensional arrays for
#                    each expectand.  The first dimension of each
#                    element indexes the Markov chains and the
#                    second dimension indexes the sequential
#                    states within each Markov chain.
# @param xlim       Optional global x-axis bounds for all pair plots.
#                   Defaults to dynamic bounds for each pair plot.
# @param ylim       Optional global y-axis bounds for all pair plots.
#                   Defaults to dynamic bounds for each pair plot.
# @param transforms An optional dictionary with expectand names for keys
#                   and transformation flags for values.  Valid flags
#                   are
#                     0: identity
#                     1: log
#                     2: logit
#                   Defaults to empty dictionary.
# @params plot_mode Optional plotting style configuration:
#                     0: Non-divergent transitions are plotted in
#                        transparent red while divergent transitions are
#                        plotted in transparent green.
#                     1: Non-divergent transitions are plotted in gray
#                        while divergent transitions are plotted in
#                        different shades of teal depending on the
#                        trajectory length.  Transitions from shorter
#                        trajectories should cluster somewhat closer to
#                        the neighborhoods with problematic geometries.
#                   Defaults to 0.
# @param max_width Maximum line width for printing
def plot_div_pairs(x_names, y_names, expectand_vals_dict,
                   diagnostics, transforms={},
                   xlim=None, ylim=None,
                   plot_mode=0, max_width=72):
  """Plot pairwise scatter plots with non-divergent and divergent
     transitions separated by color"""
  if not isinstance(x_names, list):
```

```
  raise TypeError(('Input variable `x_names` is not a list.'))

if not isinstance(y_names, list):
  raise TypeError(('Input variable `y_names` is not a list.'))

validate_dict_of_arrays(expectand_vals_dict, 'expectand_vals_dict')
validate_dict_of_arrays(diagnostics, 'diagnostics')

if not isinstance(transforms, dict):
  raise TypeError('Input variable `transforms` is not a dictionary.')

# Check transform flags
for t_name, t_value in transforms.items():
  if t_value < 0 or t_value > 2:
    desc = (f'The transform flag {t_value} for '
            f'expectand {t_name} is invalid.  '
            'Plot will default to no tranformation.')
    desc = textwrap.wrap(desc, max_width)
    print('\n'.join(desc))

# Check plot mode
if plot_mode < 0 or plot_mode > 1:
  print(f'Invalid `plot mode` value {plot_mode}.')
  return

# Transform expectand values
transformed_vals = {}

transformed_x_names = []
for name in x_names:
  try:
    t_name, t_vals = apply_transform(name,
                                     expectand_vals_dict,
                                     transforms)
  except ValueError as error:
    desc = textwrap.wrap(error, max_width)
    print('\n'.join(desc))
    return

  transformed_x_names.append(t_name)
  if t_name not in transformed_vals:
    transformed_vals[t_name] = t_vals
```

```
transformed_y_names = []
for name in y_names:
  try:
    t_name, t_vals = apply_transform(name,
                                     expectand_vals_dict,
                                     transforms)
  except ValueError as error:
    desc = textwrap.wrap(error, max_width)
    print('\n'.join(desc))

  transformed_y_names.append(t_name)
  if t_name not in transformed_vals:
    transformed_vals[t_name] = t_vals

# Create pairs of transformed expectands, dropping duplicates
pairs = []
for x_name in transformed_x_names:
  for y_name in transformed_y_names:
    if x_name == y_name:
      continue
    if [x_name, y_name] in pairs or [y_name, x_name] in pairs:
      continue
    pairs.append([x_name, y_name])

# Extract diagnostic information
divergences = diagnostics['divergent__'].flatten()

if plot_mode == 1:
  if sum(divergences) > 0:
    div_nlfs = [ x for x, d in
                 zip(diagnostics['n_leapfrog__'].flatten(),
                     divergences)
                 if d == 1  ]
    max_nlf = max(div_nlfs)
    nom_colors = [light_teal, mid_teal, dark_teal]
    cmap = LinearSegmentedColormap.from_list("teals", nom_colors,
                                             N=max_nlf)
  else:
    div_nlfs = []
    nom_colors = [light_teal, mid_teal, dark_teal]
    cmap = LinearSegmentedColormap.from_list("teals", nom_colors,
                                             N=1)
```

```
# Set plot layout dynamically
N_pairs = len(pairs)

if N_pairs == 1:
  N_cols = 1
  N_rows = 1
else:
  N_cols = 3
  N_rows = math.ceil(N_pairs / N_cols)

if N_rows > 3:
  N_rows = 3

# Plot
k = 0

for pair in pairs:
  if k == 0:
    f, axarr = plot.subplots(N_rows, N_cols, layout="constrained",
                             squeeze=False)

  x_name = pair[0]
  x_nondiv_vals = [ x for x, d in
                    zip(transformed_vals[x_name], divergences)
                    if d == 0  ]
  x_div_vals    = [ x for x, d in
                    zip(transformed_vals[x_name], divergences)
                    if d == 1  ]

  if xlim is None:
    xmin = min(numpy.concatenate((x_nondiv_vals, x_div_vals)))
    xmax = max(numpy.concatenate((x_nondiv_vals, x_div_vals)))
    local_xlim = [xmin, xmax]
  else:
    local_xlim = xlim

  y_name = pair[1]
  y_nondiv_vals = [ y for y, d in
                    zip(transformed_vals[y_name], divergences)
                    if d == 0  ]
  y_div_vals    = [ y for y, d in
                    zip(transformed_vals[y_name], divergences)
```

```
                    if d == 1  ]

  if ylim is None:
    ymin = min(numpy.concatenate((y_nondiv_vals, y_div_vals)))
    ymax = max(numpy.concatenate((y_nondiv_vals, y_div_vals)))
    local_ylim = [ymin, ymax]
  else:
    local_ylim = ylim

  idx1 = k // N_cols
  idx2 = k % N_cols

  if plot_mode == 0:
    axarr[idx1, idx2].scatter(x_nondiv_vals, y_nondiv_vals, s=5,
                              color=dark_highlight, alpha=0.05)
    axarr[idx1, idx2].scatter(x_div_vals, y_div_vals, s=5,
                              color="#00FF00", alpha=0.25)
  elif plot_mode == 1:
    axarr[idx1, idx2].scatter(x_nondiv_vals, y_nondiv_vals,
                              s=5, color="#DDDDDD")
    if len(x_div_vals) > 0:
      axarr[idx1, idx2].scatter(x_div_vals, y_div_vals, s=5,
                                cmap=cmap, c=div_nlfs)

  axarr[idx1, idx2].set_xlabel(x_name)
  axarr[idx1, idx2].set_xlim(local_xlim)
  axarr[idx1, idx2].set_ylabel(y_name)
  axarr[idx1, idx2].set_ylim(local_ylim)
  axarr[idx1, idx2].spines["top"].set_visible(False)
  axarr[idx1, idx2].spines["right"].set_visible(False)

  k += 1
  if k == N_rows * N_cols:
    # Flush current plot
    plot.show()
    k = 0

# Turn off any remaining subplots
if k > 0:
  for kk in range(k, N_rows * N_cols):
    idx1 = kk // N_cols
    idx2 = kk % N_cols
```

```
      axarr[idx1, idx2].axis('off')
    plot.show()
```

3 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.

3.1 xihat

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 will kick in.

$\hat{\xi}$ uses the tail behavior of a realized Markov chain to estimate the integrability of an expectand. More specifically $\hat{\xi}$ estimates the shape of a general Pareto density function from non-central values of the expectand.
If the tail behavior were exactly general Pareto then the larger the shape parameter ξ the fewer moments of the distribution will be well-defined. Formally the mth-order moment is well-defined only if

$$m < \frac{1}{\xi}.$$

For example with $\xi = 0.9$ the expectation $\mathbb{E}_\pi[f]$ is finite but $\mathbb{E}_\pi[f^2]$ is not. Similarly for $\xi = 0.4$ the expectations $\mathbb{E}_\pi[f]$ and $\mathbb{E}_\pi[f^2]$ are finite but the third-order moment $\mathbb{E}_\pi[f^3]$ is not.

The estimator $\hat{\xi}$ 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 $\hat{\xi}$ only estimates the tail shape I require a conservative threshold of $\hat{\xi} \geq 0.25$ 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 $\hat{\xi}$ estimator is poorly-behaved, but the boundedness also guarantees that moments of all orders exist. To make this diagnostic as robust as possible $\hat{\xi}$ will return -2 in these cases to avoid the diagnostic threshold.

```
# Compute hat{xi}, an estimate for the shape of a generalized Pareto
# distribution from a sample of positive values using the method
# introduced in "A New and Efficient Estimation Method for the
# Generalized Pareto Distribution" by Zhang and Stephens
# https://doi.org/10.1198/tech.2009.08017.
#
# Within the generalized Pareto distribution family all moments up to
# the mth order are finite if and only if
#  xi < 1 / m.
#
# @params vals A one-dimensional array of positive values.
# @return Shape parameter estimate.
def compute_xi_hat(vals):
  """Compute empirical Pareto shape configuration for a positive sample"""
  N = len(vals)
  sorted_vals = sorted(vals)

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

  if (sorted_vals[0] < 0):
    print("Sequence values must be positive.")
    return math.nan

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

  # Heuristic 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_vals[-1] \
                   + (1 - math.sqrt(M / (m + 0.5))) / (3 * q)
    if b_hat_vec[m] != 0:
      xi_hat = numpy.mean( [ math.log(1 - b_hat_vec[m] * f)
                             for f in sorted_vals ] )
      log_w_vec[m] = N * (   math.log(-b_hat_vec[m] / xi_hat)
```

```
                            - xi_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_vals ] )
```

```
# Compute empirical generalized Pareto shape for upper and lower tails
# for an arbitrary sample of expectand values, ignoring any
# autocorrelation between the values.
# @param vals A one-dimensional array of expectand values.
# @return Left and right shape estimators.
def compute_tail_xi_hats(vals):
  """Compute empirical Pareto shape configuration for upper and lower tails"""
  v_center = numpy.median(vals)

  # Isolate lower and upper tails which can be adequately modeled by a
  # generalized Pareto shape for sufficiently well-behaved distributions
  vals_left = [ math.fabs(v - v_center) for v in vals if v <= v_center ]
  N = len(vals_left)
  M = int(min(0.2 * N, 3 * 3 * math.sqrt(N)))
  vals_left = vals_left[M:N]

  vals_right = [ v - v_center for v in vals if v > v_center ]
  N = len(vals_right)
  M = int(min(0.2 * N, 3 * 3 * math.sqrt(N)))
  vals_right = vals_right[M:N]

  # Default to NaN if left tail is ill-defined
  xi_hat_left = math.nan
  if len(vals_left) > 40:
    xi_hat_left = compute_xi_hat(vals_left)
```

```
  # Default to NaN if right tail is ill-defined
  xi_hat_right = math.nan
  if len(vals_right) > 40:
    xi_hat_right = compute_xi_hat(vals_right)

  return [xi_hat_left, xi_hat_right]
```

```
# Check upper and lower tail behavior of a given expectand output
# ensemble.
# @param expectand_vals A two-dimensional array of expectand values with
#                       the first dimension indexing the Markov chains
#                       and the second dimension indexing the sequential
#                       states within each Markov chain.
# @param max_width Maximum line width for printing
def check_tail_xi_hats(expectand_vals, max_width=72):
  """Check empirical Pareto shape configuration for upper and lower
     tails of a given expectand output ensemble"""
  validate_array(expectand_vals, 'expectand_vals')

  C = expectand_vals.shape[0]
  no_warning = True

  for c in range(C):
    xi_hats = compute_tail_xi_hats(expectand_vals[c,:])
    xi_hat_threshold = 0.25
    if math.isnan(xi_hats[0]) and math.isnan(xi_hats[1]):
      no_warning = False
      print(f'  Chain {c + 1}: Both left and right tail '
            'hat{{xi}}s are Nan.\n')
    elif math.isnan(xi_hats[0]):
      no_warning = False
      print(f'  Chain {c + 1}: Left tail '
            'hat{{xi}} is Nan.\n')
    elif math.isnan(xi_hats[1]):
      no_warning = False
      print(f'  Chain {c + 1}: Right tail '
            'hat{{xi}} is Nan.\n')
    elif (    xi_hats[0] >= xi_hat_threshold
         and xi_hats[1] >= xi_hat_threshold):
      no_warning = False
      print(f'  Chain {c + 1}: Both left and right tail '
            f'hat{{xi}}s ({xi_hats[0]:.3f}, '
```

```
            f'{xi_hats[1]:.3f}) exceed '
            f'{xi_hat_threshold}.\n')
    elif (    xi_hats[0] < xi_hat_threshold
          and xi_hats[1] >= xi_hat_threshold):
      no_warning = False
      print(f'  Chain {c + 1}: Right tail hat{{xi}} '
            f'({xi_hats[1]:.3f}) exceeds '
            f'{xi_hat_threshold}.\n')
    elif (    xi_hats[0] >= xi_hat_threshold
          and xi_hats[1] < xi_hat_threshold):
      no_warning = False
      print(f'  Chain {c + 1}: Left tail hat{{xi}} '
            f'({xi_hats[0]:.3f}) exceeds '
            f'{xi_hat_threshold}.\n')

  if no_warning:
    print('Expectand appears to be sufficiently integrable.\n')
  else:
    desc = ('  Large tail xi_hats suggest that the expectand might'
            'not be sufficiently integrable.')
    desc = textwrap.wrap(desc, max_width)
    desc.append(' ')
    print('\n'.join(desc))
```

3.2 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.

```
# Compute empirical mean and variance of a given sequence with a single
# pass using Welford accumulators.
# @params vals A one-dimensional array of sequential expectand values.
# @return The empirical mean and variance.
def welford_summary(vals):
  """Welford accumulator for empirical mean and variance of a
     given sequence"""
```

```
  mean = 0
  var = 0

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

  var /= (len(vals) - 1)

  return [mean, var]
```

```
# Check expectand output ensemble for vanishing empirical variance.
# @param expectand_vals A two-dimensional array of expectand values with
#                       the first dimension indexing the Markov chains
#                       and the second dimension indexing the sequential
#                       states within each Markov chain.
# @param max_width Maximum line width for printing
def check_variances(expectand_vals, max_width=72):
  """Check expectand output ensemble for vanishing empirical variance"""
  validate_array(expectand_vals, 'expectand_vals')

  C = expectand_vals.shape[0]
  no_warning = True

  for c in range(C):
    var = welford_summary(expectand_vals[c,:])[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.\n')
  else:
    desc = ('  If the expectand is not expected (haha) to be '
            'constant then the Markov transitions are misbehaving.')
    desc = textwrap.wrap(desc, max_width)
    desc.append(' ')
    print('\n'.join(desc))
```

3.3 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.

```
# Split a sequence of expectand values in half to create an initial and
# terminal Markov chains
# @params chain A sequence of expectand values derived from a single
#               Markov chain.
# @return Two subsequences of expectand values.
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] ]
```

```
# Compute split hat{R} for the expectand values across a Markov chain
# ensemble.
# @param expectand_vals A two-dimensional array of expectand values with
#                       the first dimension indexing the Markov chains
#                       and the second dimension indexing the sequential
#                       states within each Markov chain.
# @return Split Rhat estimate.
def compute_split_rhat(expectand_vals):
  """Compute split hat{R} for an expectand output ensemble across
     a collection of Markov chains"""
  validate_array(expectand_vals, 'expectand_vals')

  split_chain_vals = [ c for chain_vals in expectand_vals
```

```
                        for c in split_chain(chain_vals) ]
  N_chains = len(split_chain_vals)
  N = sum([ len(vals) for vals in split_chain_vals ])

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

  for c, vals in enumerate(split_chain_vals):
    summary = welford_summary(vals)
    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
```

```
# Compute split hat{R} for all input expectands
# @param expectand_vals_dict A dictionary of two-dimensional arrays for
#                            each expectand.  The first dimension of
#                            each element indexes the Markov chains and
#                            the second dimension indexes the sequential
#                            states within each Markov chain.
def compute_split_rhats(expectand_vals_dict):
  """Compute split hat{R} for all expectand output ensembles across
     a collection of Markov chains"""
  validate_dict_of_arrays(expectand_vals_dict, 'expectand_vals_dict')

  rhats = []
  for name in expectand_vals_dict:
    expectand_vals = expectand_vals_dict[name]
    rhats.append(compute_split_rhat(expectand_vals))

  return rhats
```

```
# Check split hat{R} across a given expectand output ensemble.
# @param expectand_vals A two-dimensional array of expectand values with
#                       the first dimension indexing the Markov chains
#                       and the second dimension indexing the sequential
#                       states within each Markov chain.
# @param max_width Maximum line width for printing
def check_rhat(expectand_vals, max_width=72):
  """Check split hat{R} for all expectand output ensembles across
     a collection of Markov chains"""
  validate_array(expectand_vals, 'expectand_vals')

  rhat = compute_split_rhat(expectand_vals)

  no_warning = True

  if math.isnan(rhat):
    print('All Markov chains appear to be frozen.')
  elif rhat > 1.1:
    print(f'Split hat{{R}} is {rhat:.3f}.')
    no_warning = False

  if no_warning:
    desc = ('Markov chain behavior is consistent with equilibrium.')
    desc = textwrap.wrap(desc, max_width)
    desc.append(' ')
    print('\n'.join(desc))
  else:
    desc = ('Split hat{R} larger than 1.1 suggests that at least one '
            'of the Markov chains has not reached an equilibrium.')
    desc = textwrap.wrap(desc, max_width)
    desc.append(' ')
    print('\n'.join(desc))
```

3.4 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 $\hat{\tau}[f]$.

```
# Compute empirical integrated autocorrelation time, \hat{tau}, for a
# sequence of expectand values.
# @param vals A one-dimensional array of expectand values.
# @return Left and right shape estimators.
def compute_tau_hat(vals):
  """Compute empirical integrated autocorrelation time for a sequence"""
  # Compute empirical autocorrelations
  N = len(vals)
  m, v = welford_summary(vals)
  zs = [ val - m for val in vals ]

  if v < 1e-10:
    return math.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
```

This, estimate, however, can be unreliable if the Markov chains have not had sufficient time to explore. In my experience a good rule of thumb is that the empirical integrated autocorrelation time has cannot be larger than five times the number of total iterations,

$$\hat{\tau}[f] < 5 \cdot N.$$

Equivalently the incremental empirical integrated autocorrelation time cannot be larger than five,

$$\frac{\hat{\tau}[f]}{N} < 5.$$

```
# Check the incremental empirical integrated autocorrelation time for
# all the given expectand values.
# @param expectand_vals A two-dimensional array of expectand values with
#                       the first dimension indexing the Markov chains
#                       and the second dimension indexing the sequential
#                       states within each Markov chain.
# @param max_width Maximum line width for printing
def check_inc_tau_hat(expectand_vals, max_width=72):
  """Check that the incremental empirical integrated autocorrelation
     time of the given expectand values is sufficiently large."""
  validate_array(expectand_vals, 'expectand_vals')

  no_warning = True
  C = expectand_vals.shape[0]
  S = expectand_vals.shape[1]

  for c in range(C):
```

```
    tau_hat = compute_tau_hat(expectand_vals[c,:])
    inc_tau_hat_per = tau_hat / S
    if tau_hat_per > 5:
      print(f'Chain {c + 1}: The incremental empirical integrated '
            f'autocorrelation time {inc_tau_hat :.3f} is too large.')
      no_warning = False

  if no_warning:
    desc = ('The incremental empirical integrated autocorrelation '
            'time is small enough for the empirical autocorrelation '
            'estimates to be reliable.')
    desc = textwrap.wrap(desc, max_width)
    desc.append(' ')
    print('\n'.join(desc))
  else:
    desc = ('If the incremental empirical integrated autocorrelation '
            'times are too large then the Markov '
            'chains have not explored long enough for the '
            'autocorrelation estimates to be reliable.')
    desc = textwrap.wrap(desc, max_width)
    desc.append(' ')
    print('\n'.join(desc))
```

The integrated autocorrelation times moderates the asymptotic variance of well-behaved Markov chain Monte Carlo estimators through the effective sample size,

$$\mathrm{ESS}[f] = \frac{N}{\tau[f]},$$

or in practice the empirical effective sample size that we estimate from the realized Markov chains,

$$\hat{\mathrm{ESS}}[f] = \frac{N}{\hat{\tau}[f]}.$$

The effective sample size can be interpreted as how large of an ensemble of exact samples we would need to achieve the same estimator error for the particular expectand of interest.

```
# Compute the minimum empirical effective sample size, or \hat{ESS},
# across the Markov chains for the given expectands.
# @param expectand_vals_dict A dictionary of two-dimensional arrays for
#                            each expectand.  The first dimension of
#                            each element indexes the Markov chains and
#                            the second dimension indexes the sequential
```

```
#                             states within each Markov chain.
def compute_min_ess_hats(expectand_vals_dict):
  """Compute the minimum empirical integrated autocorrelation time
     across a collection of Markov chains for all expectand output
     ensembles"""
  validate_dict_of_arrays(expectand_vals_dict, 'expectand_vals_dict')

  min_ess_hats = []
  for name in expectand_vals_dict:
    expectand_vals = expectand_vals_dict[name]
    C = expectand_vals.shape[0]
    S = expectand_vals.shape[0]

    ess_hats = [None] * 4
    for c in range(C):
      tau_hat = compute_tau_hat(expectand_vals[c,:])
      ess_hats[c] = S / tau_hat

    min_ess_hats.append(min(ess_hats))

  return min_ess_hats
```

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

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

The necessary 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 more than sufficient for most applications and provides a useful rule of thumb. In some applications even smaller effective sample sizes can yield sufficiently precise Markov chain Monte Carlo estimators.

```
# Check the empirical effective sample size \hat{ESS} for the given
# expectand values.
# @param expectand_vals A two-dimensional array of expectand values with
#                       the first dimension indexing the Markov chains
#                       and the second dimension indexing the sequential
#                       states within each Markov chain.
# @param min_ess_hat_per_chain The minimum empirical effective sample
#                              size before a warning message is passed.
```

```
# @param max_width Maximum line width for printing
def check_ess_hat(expectand_vals,
                  min_ess_hat_per_chain=100,
                  max_width=72):
  """Check the empirical effective sample size for all expectand
     output ensembles"""
  validate_array(expectand_vals, 'expectand_vals')

  no_warning = True
  C = expectand_vals.shape[0]
  S = expectand_vals.shape[1]

  for c in range(C):
    tau_hat = compute_tau_hat(expectand_vals[c,:])
    ess_hat = S / tau_hat
    if ess_hat < min_ess_hat_per_chain:
      print(f'Chain {c + 1}: The empirical effective sample size '
            f'{ess_hat :.1f} is too small.')
      no_warning = False

  if no_warning:
    desc = ('Assuming that a central limit theorem holds the '
            'empirical effective sample size is large enough '
            'for Markov chain Monte Carlo estimation to be'
            'reasonably precise.')
    desc = textwrap.wrap(desc, max_width)
    desc.append(' ')
    print('\n'.join(desc))
  else:
    desc = ('Small empirical effective sample sizes result in '
            'imprecise Markov chain Monte Carlo estimators.')
    desc = textwrap.wrap(desc, max_width)
    desc.append(' ')
    print('\n'.join(desc))
```

For example empirical effective sample sizes can provide a useful way to distinguish if some diagnostic failures are due to Markov chains that are just too short or more persistent problems.

3.5 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-specific diagnostics.
# @param expectand_vals_dict A dictionary of two-dimensional arrays for
#                            each expectand.  The first dimension of
#                            each element indexes the Markov chains and
#                            the second dimension indexes the sequential
#                            states within each Markov chain.
# @param min_ess_hat_per_chain The minimum empirical effective sample
#                              size before a warning message is passed.
# @param exclude_zvar Binary variable to exclude all expectands with
#                     vanishing empirical variance from other diagnostic
#                     checks.
# @param max_width Maximum line width for printing
def check_all_expectand_diagnostics(expectand_vals_dict,
                                    min_ess_hat_per_chain=100,
                                    exclude_zvar=False,
                                    max_width=72):
  """Check all expectand diagnostics"""
  validate_dict_of_arrays(expectand_vals_dict, 'expectand_vals_dict')

  no_xi_hat_warning = True
  no_zvar_warning = True
  no_rhat_warning = True
  no_inc_tau_hat_warning = True
  no_ess_hat_warning = True

  message = ""

  for name in expectand_vals_dict:
    expectand_vals = expectand_vals_dict[name]
    C = expectand_vals.shape[0]
    S = expectand_vals.shape[1]

    local_warning = False
    local_message = name + ':\n'

    if exclude_zvar:
      # Check zero variance across all Markov chains for exclusion
      any_zvar = False
```

```
    for c in range(C):
      var = welford_summary(expectand_vals[c,:])[1]
      if var < 1e-10:
        any_zvar = True

    if any_zvar:
      continue

  for c in range(C):
    # Check tail xi_hats in each Markov chain
    xi_hats = compute_tail_xi_hats(expectand_vals[c,:])
    xi_hat_threshold = 0.25
    if math.isnan(xi_hats[0]) and math.isnan(xi_hats[1]):
      no_xi_hat_warning = False
      local_warning = True
      local_message += (f'  Chain {c + 1}: Both left and right tail '
                        'hat{{xi}}s are Nan.\n')
    elif math.isnan(xi_hats[0]):
      no_xi_hat_warning = False
      local_warning = True
      local_message += (f'  Chain {c + 1}: Left tail '
                        'hat{{xi}} is Nan.\n')
    elif math.isnan(xi_hats[1]):
      no_xi_hat_warning = False
      local_warning = True
      local_message += (f'  Chain {c + 1}: Right tail '
                        'hat{{xi}} is Nan.\n')
    if (    xi_hats[0] >= xi_hat_threshold
        and xi_hats[1] >= xi_hat_threshold):
      no_xi_hat_warning = False
      local_warning = True
      local_message += (f'  Chain {c + 1}: Both left and right tail '
                        f'hat{{xi}}s ({xi_hats[0]:.3f}, '
                        f'{xi_hats[1]:.3f}) exceed '
                        f'{xi_hat_threshold}.\n')
    elif (    xi_hats[0] < xi_hat_threshold
          and xi_hats[1] >= xi_hat_threshold):
      no_xi_hat_warning = False
      local_warning = True
      local_message += (f'  Chain {c + 1}: Right tail hat{{xi}} '
                        f'({xi_hats[1]:.3f}) exceeds '
                        f'{xi_hat_threshold}.\n')
```

```
    elif (    xi_hats[0] >= xi_hat_threshold
          and xi_hats[1] < xi_hat_threshold):
      no_xi_hat_warning = False
      local_warning = True
      local_message += (f'  Chain {c + 1}: Left tail hat{{xi}} '
                        f'({xi_hats[0]:.3f}) exceeds '
                        f'{xi_hat_threshold}.\n')

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

  # Check split Rhat across Markov chains
  rhat = compute_split_rhat(expectand_vals)

  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):
    tau_hat = compute_tau_hat(expectand_vals[c,:])

    # Check incremental empirical integrated autocorrelation time
    inc_tau_hat = tau_hat / S
    if inc_tau_hat > 5:
      no_inc_tau_hat_warning = False
      local_warning = True
      local_message += (f'  Chain {c + 1}: Incremental hat{{tau}} '
                        f'({inc_tau_hat:.1f}) is too large.\n')
    # Check empirical effective sample size
    ess_hat = S / tau_hat
    if ess_hat < min_ess_hat_per_chain:
      no_ess_hat_warning = False
      local_warning = True
      local_message += (f'  Chain {c + 1}: hat{{ESS}} ({ess_hat:.1f}) '
```

```
                        'is smaller than desired '
                        f'({min_ess_hat_per_chain:.0f}).\n')

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

if (    no_xi_hat_warning and no_zvar_warning
    and no_rhat_warning   and no_inc_tau_hat_warning
    and no_ess_hat_warning):
  desc = ('All expectands checked appear to be behaving well enough '
          'for reliable Markov chain Monte Carlo estimation.')
  desc = textwrap.wrap(desc, max_width)
  desc.append(' ')
  print('\n'.join(desc))
  return

print(message)

if not no_xi_hat_warning:
  desc = ('Large tail hat{xi}s suggest that the expectand '
          'might not be sufficiently integrable.')
  desc = textwrap.wrap(desc, max_width)
  desc.append(' ')
  print('\n'.join(desc))

if not no_zvar_warning:
  desc = ('If the expectands are not constant then zero empirical '
          'variance suggests that the Markov transitions may be '
          'misbehaving.')
  desc = textwrap.wrap(desc, max_width)
  desc.append(' ')
  print('\n'.join(desc))

if not no_rhat_warning:
  desc = ('Split Rhat larger than 1.1 suggests that at least one of '
          'the Markov chains has not reached an equilibrium.')
  desc = textwrap.wrap(desc, max_width)
  desc.append(' ')
  print('\n'.join(desc))

if not no_inc_tau_hat_warning:
  desc = ('If the incremental empirical integrated autocorrelation '
```

```
            'times are too large then the Markov '
            'chains have not explored long enough for the '
            'autocorrelation estimates to be reliable.')
    desc = textwrap.wrap(desc, max_width)
    desc.append(' ')
    print('\n'.join(desc))

  if not no_ess_hat_warning:
    desc = ('Small empirical effective sample sizes result in '
            'imprecise Markov chain Monte Carlo estimators.')
    desc = textwrap.wrap(desc, max_width)
    desc.append(' ')
    print('\n'.join(desc))

  return
```

That said for particularly problematic fits the output from checking all of the expectands can be overwhelming. In cases where that may be a risk we can summarize the output more compactly.

```
# Summary all expectand-specific diagnostics.
# @param expectand_vals_dict A dictionary of two-dimensional arrays for
#                            each expectand.  The first dimension of
#                            each element indexes the Markov chains and
#                            the second dimension indexes the sequential
#                            states within each Markov chain.
# @param min_ess_hat_per_chain The minimum empirical effective sample
#                              size before a warning message is passed.
# @param exclude_zvar Binary variable to exclude all expectands with
#                     vanishing empirical variance from other diagnostic
#                     checks.
# @param max_width Maximum line width for printing
def summarize_expectand_diagnostics(expectand_vals_dict,
                                    min_ess_hat_per_chain=100,
                                    exclude_zvar=False,
                                    max_width=72):
  """Summarize expectand diagnostics"""
  validate_dict_of_arrays(expectand_vals_dict, 'expectand_vals_dict')

  failed_names = []
  failed_xi_hat_names = []
  failed_zvar_names = []
```

```
failed_rhat_names = []
failed_inc_tau_hat_names = []
failed_ess_hat_names = []

for name in expectand_vals_dict:
  expectand_vals = expectand_vals_dict[name]
  C = expectand_vals.shape[0]
  S = expectand_vals.shape[1]

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

  for c in range(C):
    # Check tail xi_hats in each Markov chain
    xi_hats = compute_tail_xi_hats(expectand_vals[c,:])
    xi_hat_threshold = 0.25
    if math.isnan(xi_hats[0]) or math.isnan(xi_hats[1]):
      failed_names.append(name)
      failed_xi_hat_names.append(name)
    if (   xi_hats[0] >= xi_hat_threshold
        or xi_hats[1] >= xi_hat_threshold):
      failed_names.append(name)
      failed_xi_hat_names.append(name)

    # Check empirical variance in each Markov chain
    var = welford_summary(expectand_vals[c,:])[1]
    if var < 1e-10:
      failed_names.append(name)
      failed_zvar_names.append(name)

  # Check split Rhat across Markov chains
  rhat = compute_split_rhat(expectand_vals)

  if math.isnan(rhat):
    failed_names.append(name)
```

```
      failed_rhat_names.append(name)
    elif rhat > 1.1:
      failed_names.append(name)
      failed_rhat_names.append(name)

    for c in range(C):
      tau_hat = compute_tau_hat(expectand_vals[c,:])

      # Check incremental empirical integrated autocorrelation time
      inc_tau_hat = tau_hat / S
      if inc_tau_hat > 5:
        failed_names.append(name)
        failed_inc_tau_hat_names.append(name)

      # Check empirical effective sample size
      ess_hat = S / tau_hat

      if ess_hat < min_ess_hat_per_chain:
        failed_names.append(name)
        failed_ess_hat_names.append(name)

failed_names = list(numpy.unique(failed_names))
if len(failed_names):
  desc = (f'The expectands {", ".join(failed_names)} '
          'triggered diagnostic warnings.')
  desc = textwrap.wrap(desc, max_width)
  desc.append(' ')
  print('\n'.join(desc))
else:
  desc = ('All expectands checked appear to be behaving well enough '
          'for reliable Markov chain Monte Carlo estimation.')
  desc = textwrap.wrap(desc, max_width)
  desc.append(' ')
  print('\n'.join(desc))

failed_xi_hat_names = list(numpy.unique(failed_xi_hat_names))
if len(failed_xi_hat_names):
  desc = (f'The expectands {", ".join(failed_xi_hat_names)} '
          'triggered tail hat{xi} warnings.')
  desc = textwrap.wrap(desc, max_width)
  print('\n'.join(desc))
```

```
  desc = ('  Large tail hat{xi}s suggest that the expectand '
          'might not be sufficiently integrable.')
  desc = textwrap.wrap(desc, max_width)
  desc.append(' ')
  print('\n'.join(desc))

failed_zvar_names = list(numpy.unique(failed_zvar_names))
if len(failed_zvar_names):
  desc = (f'The expectands {", ".join(failed_zvar_names)} '
           'triggered zero variance warnings.')
  desc = textwrap.wrap(desc, max_width)
  print('\n'.join(desc))

  desc = ('  If the expectands are not constant then zero empirical'
          ' variance suggests that the Markov'
          ' transitions may be misbehaving.')
  desc = textwrap.wrap(desc, max_width)
  desc.append(' ')
  print('\n'.join(desc))

failed_rhat_names = list(numpy.unique(failed_rhat_names))
if len(failed_rhat_names):
  desc = (f'The expectands {", ".join(failed_rhat_names)} '
           'triggered hat{R} warnings.')
  desc = textwrap.wrap(desc, max_width)
  print('\n'.join(desc))

  desc = ('  Split Rhat larger than 1.1 suggests that at '
          'least one of the Markov chains has not reached '
          'an equilibrium.')
  desc = textwrap.wrap(desc, max_width)
  desc.append(' ')
  print('\n'.join(desc))

failed_inc_tau_hat_names = list(numpy.unique(failed_inc_tau_hat_names))
if len(failed_inc_tau_hat_names):
  desc = (f'The expectands {", ".join(failed_rhat_names)} '
           'triggered incremental hat{tau} warnings.')
  desc = textwrap.wrap(desc, max_width)
  print('\n'.join(desc))

  desc = ('If the incremental empirical integrated autocorrelation '
```

```
            'times per iteration are too large then the Markov '
            'chains have not explored long enough for the '
            'autocorrelation estimates to be reliable.')
    desc = textwrap.wrap(desc, max_width)
    desc.append(' ')
    print('\n'.join(desc))

  failed_ess_hat_names = list(numpy.unique(failed_ess_hat_names))
  if len(failed_ess_hat_names):
    desc = (f'The expectands {", ".join(failed_ess_hat_names)} '
             'triggered hat{ESS} warnings.')
    desc = textwrap.wrap(desc, max_width)
    print('\n'.join(desc))

    desc = ('Small empirical effective sample sizes result in '
            'imprecise Markov chain Monte Carlo estimators.')
    desc = textwrap.wrap(desc, max_width)
    desc.append(' ')
    print('\n'.join(desc))
```

Alternatively we might filter the expectands, keeping only those of immediate interest.

```
# Filter `expectand_vals_dict` by name.
# @param expectand_vals_dict A dictionary of two-dimensional arrays for
#                            each expectand.  The first dimension of
#                            each element indexes the Markov chains and
#                            the second dimension indexes the sequential
#                            states within each Markov chain.
# @param requested_names List of expectand names to keep.
# @param check_arrays Binary variable indicating whether or not
#                     requested names should be expanded to array
#                     components.
# @param max_width Maximum line width for printing
# @return A dictionary of two-dimensional arrays for each requested
#         expectand.
def filter_expectands(expectand_vals_dict, requested_names,
                      check_arrays=False, max_width=72):
  validate_dict_of_arrays(expectand_vals_dict, 'expectand_vals_dict')

  if len(requested_names) == 0:
    raise ValueError('Input variable `requested_names` '
                     'must be non-empty.')
```

```
if check_arrays is True:
  good_names = []
  bad_names = []
  for name in requested_names:
    # Search for array suffix
    array_names = [ key for key in expectand_vals_dict.keys()
                    if re.match('^' + name + '\[', key) ]
    # Append array names, if found
    if len(array_names) > 0:
      good_names += array_names
    else:
      if name in expectand_vals_dict.keys():
        # Append bare name, if found
        good_names.append(name)
      else:
        # Add to list of bad names
        bad_names.append(name)
else:
  bad_names = \
    set(requested_names).difference(expectand_vals_dict.keys())
  good_names = \
    set(requested_names).intersection(expectand_vals_dict.keys())

if len(bad_names) > 0:
  message = (f'The expectands {", ".join(bad_names)} '
             'were not found in the `expectand_vals_dict` '
             'object and will be ignored.\n\n')
  message = textwrap.wrap(message, max_width)
  message.append(' ')
  print('\n'.join(message))

return { name: expectand_vals_dict[name] for name in good_names }
```

3.6 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 empirical autocorrelations for a given Markov chain sequence
# @parmas vals A one-dimensional array of sequential expectand values.
# @return A one-dimensional array of empirical autocorrelations at each
#         lag up to the length of the sequence.
def compute_rhos(vals):
  """Visualize empirical autocorrelations for a given sequence"""
  # Compute empirical autocorrelations
  N = len(vals)
  m, v = welford_summary(vals)
  zs = [ val - m for val in vals ]

  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
```

```
# Plot empirical correlograms for a given expectand across a Markov
# chain ensemble.
# @ax Matplotlib axis object
# @param expectand_vals A two-dimensional array of expectand values with
#                       the first dimension indexing the Markov chains
#                       and the second dimension indexing the sequential
#                       states within each Markov chain.
# @param max_L Maximum autocorrelation lag
# @param rho_lim Plotting range of autocorrelation values
# @display_name Name of expectand
def plot_empirical_correlogram(ax,
                               expectand_vals,
                               max_L,
                               rho_lim=[-0.2, 1.1],
                               name=""):
  """Plot empirical correlograms for a given expectand across a Markov
     chain ensemble."""
  validate_array(expectand_vals, 'expectand_vals')

  C = expectand_vals.shape[0]

  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]

  for c in range(C):
```

```
    rhos = compute_rhos(expectand_vals[c,:])
    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(rho_lim[0], rho_lim[1])
  ax.spines["top"].set_visible(False)
  ax.spines["right"].set_visible(False)
```

3.7 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.

```
# Visualize the projection of a Markov chain ensemble along two
# expectands as a pairs plot.  Point colors darken along each Markov
# chain to visualize the autocorrelation.
# @param expectand1_vals A two-dimensional array of expectand values
#                        with the first dimension indexing the Markov
#                        chains and the second dimension indexing the
#                        sequential states within each Markov chain.
# @params display_name1 Name of first expectand
# @param expectand2_vals A two-dimensional array of expectand values
#                        with the first dimension indexing the Markov
#                        chains and the second dimension indexing the
#                        sequential states within each Markov chain
# @params display_name2 Name of second expectand
def plot_pairs_by_chain(expectand1_vals, display_name1,
                        expectand2_vals, display_name2):
  """Plot two expectand output ensembles against each other separated by
     Markov chain """
  validate_array(expectand1_vals, 'expectand1_vals')
  C1 = expectand1_vals.shape[0]
  S1 = expectand1_vals.shape[1]
```

```
validate_array(expectand2_vals, 'expectand2_vals')
C2 = expectand2_vals.shape[0]
S2 = expectand2_vals.shape[1]

if C1 != C2:
  C = min(C1, C2)
  C1 = C
  C2 = C
  print(f'Plotting only {C} Markov chains.')

if S1 != S2:
  S = min(S1, S2)
  S1 = S
  S2 = S
  print(f'Plotting only {S} samples per Markov chain.')

colors = ["#DCBCBC", "#C79999", "#B97C7C",
          "#A25050", "#8F2727", "#7C0000"]
cmap = LinearSegmentedColormap.from_list("reds", colors, N=S1)

min_x = min(expectand1_vals.flatten())
max_x = max(expectand1_vals.flatten())

min_y = min(expectand2_vals.flatten())
max_y = max(expectand2_vals.flatten())

N_plots = C1
N_cols = 2
N_rows = math.ceil(N_plots / N_cols)
f, axarr = plot.subplots(N_rows, N_cols, layout="constrained")
k = 0

for c in range(C1):
  idx1 = k // N_cols
  idx2 = k % N_cols
  k += 1

  axarr[idx1, idx2].scatter(expectand1_vals.flatten(),
                            expectand2_vals.flatten(),
                            color="#DDDDDD", s=5, zorder=3)
  axarr[idx1, idx2].scatter(expectand1_vals[c,:], expectand2_vals[c,:],
                            cmap=cmap, c=range(S1), s=5, zorder=4)
```

```
    axarr[idx1, idx2].set_title(f'Chain {c + 1}')
    axarr[idx1, idx2].set_xlabel(display_name1)
    axarr[idx1, idx2].set_xlim([min_x, max_x])
    axarr[idx1, idx2].set_ylabel(display_name2)
    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.show()
```

4 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.

When interested in expectands that have not already been computed we will need to evaluate the existing samples on these new functions, generating pushforward samples.

```
# Evaluate an expectand on the values of a one-dimensional input
# variable.
# @param input_vals A two-dimensional array of expectand values with
#                   the first dimension indexing the Markov chains
#                   and the second dimension indexing the sequential
#                   states within each Markov chain.
# @param expectand Scalar function to be applied to the Markov chain
#                  states.
# @return A two-dimensional array of expectand values with the
#         first dimension indexing the Markov chains and the
#         second dimension indexing the sequential states within
#         each Markov chain.
def eval_uni_expectand_pushforward(input_vals, expectand):
  """Evaluate an expectand along a Markov chain"""
  return numpy.vectorize(expectand)(input_vals)
```

```
# Create a nested list of element names, including indexing information,
# from the specified dimensions.  For example `name_array('x', [2, 3]))`
# returns the nested list
# >> [['x[1,1]', 'x[1,2]', 'x[1,3]'],
# >>  ['x[2,1]', 'x[2,2]', 'x[2,3]']]
# @ param base Base name.
```

```
# @ param dims List of array dimensions.
# @ param current_idxs Dimensions at current level of recursion.
# @ return Array of element names with dimensions given by dims.
def name_nested_list(base, dims, current_idxs=[]):
  next_dim = len(current_idxs)
  if next_dim == len(dims):
    str_idxs = ','.join([ str(idx + 1) for idx in current_idxs ])
    return f'{base}[{str_idxs}]'
  else:
    return [ name_nested_list(base, dims, current_idxs + [d])
             for d in range(dims[next_dim]) ]
```

```
# Create a numpy array of element names from the specified dimensions.
# For example `name_array('x', [2, 3]))` returns the array
# >> array([['x[1,1]', 'x[1,2]', 'x[1,3]'],
# >>        ['x[2,1]', 'x[2,2]', 'x[2,3]']], dtype='<U6')
# @ param base Base name.
# @ param dims Vector of array dimensions.
# @ return Array of element names with dimensions given by dims.
def name_array(base, dims):
  # Validate inputs
  if not isinstance(base, str):
    raise TypeError(f'Input variable {base} is not a string.')

  if not isinstance(dims, list):
    raise TypeError(f'Input variable {dims} is not a list.')

  if not all([ isinstance(d, int) for d in dims ]):
    raise TypeError(f'The elements of input variable {dims} '
                    'are not all integers.')

  # Create element names and format them into desired array.
  return numpy.array(name_nested_list(base, dims))
```

```
# Evaluate an expectand on the values of an arbitrary number of input
# variables.  Expectand must return a single float, int, or logical
# output.
#
# By default expectand argument values are accessed by name
# in expectand_vals_dict.  If a non-null alt_arg_names is provided then
# the alternate names are used to access values in expectand_vals_dict.
# The elements of alt_arg_names can also be string arrays of arbitrary
```

```
# dimension in which case the individual element values are first
# accessed then formatted into matching numeric arrays before being
# passed to the expectand.
#
# @param expectand_vals_dict A dictionary of two-dimensional arrays for
#                            each expectand.  The first dimension of
#                            each element indexes the Markov chains and
#                            the second dimension indexes the sequential
#                            states within each Markov chain.
# @param expectand Expectand with arbitrary input space.
# @param alt_arg_names Optional named list of alternate names for the
#                      nominal expectand argument names; when used all
#                      expectand argument names must be included.
def eval_expectand_pushforward(expectand_vals_dict,
                               expectand,
                               alt_arg_names=None):
  """Evaluate an expectand on the values of an arbitrary number
     of input variables."""
  # Validate inputs
  validate_dict_of_arrays(expectand_vals_dict, 'expectand_vals_dict')

  if not callable(expectand):
    raise TypeError('Input variable `expectand` is not a '
                    'callable function.')

  if alt_arg_names is not None:
    if not isinstance(alt_arg_names, dict):
      raise TypeError('Input variable `alt_arg_names` '
                      'is not a dictionary.')

  # Check existence of all expectand arguments
  spec = inspect.getfullargspec(expectand)
  nominal_arg_names = spec.args

  if alt_arg_names is None:
    check_arg_names = nominal_arg_names
  else:
    alt_keys = alt_arg_names.keys()
    missing_args = set(nominal_arg_names).difference(alt_keys)

    if len(missing_args) == 1:
      raise ValueError( 'The nominal expectand argument '
```

```
                      f'{", ".join(missing_args)} does not have '
                       'a replacement in `alt_arg_names`.')
  elif len(missing_args) > 1:
    raise ValueError( 'The nominal expectand arguments '
                      f'{", ".join(missing_args)} do not have '
                       'replacements in `alt_arg_names`.')

  arglistlist = [ v.flatten().tolist()
                  if isinstance(v, numpy.ndarray)
                  else [v]
                  for k, v in alt_arg_names.items() ]
  check_arg_names = [ arg for arglist in arglistlist
                      for arg in arglist ]

missing_args = \
  set(check_arg_names).difference(expectand_vals_dict.keys())
if len(missing_args) == 1:
  raise ValueError( 'The expectand argument '
                    f'{", ".join(missing_args)} is not in '
                     '`expectand_vals_dict`.')
elif len(missing_args) > 1:
  raise ValueError( 'The expectand arguments '
                    f'{", ".join(missing_args)} are not in '
                     '`expectand_vals_dict`.')

# Apply expectand to all inputs
C = next(iter(expectand_vals_dict.values())).shape[0]
S = next(iter(expectand_vals_dict.values())).shape[1]

pushforward_vals = numpy.zeros([C, S])
for c in range(C):
  for s in range(S):
    def access_val(name):
      return expectand_vals_dict[name][c, s]

    if alt_arg_names is None:
      arg_vals = [ access_val(name)
                   for name in nominal_arg_names ]
    else:
      arg_vals = [ numpy.vectorize(access_val)(alt_arg_names[name])
                   if isinstance(alt_arg_names[name], numpy.ndarray)
                   else access_val(alt_arg_names[name])
```

```
                    for name in nominal_arg_names ]

      pushforward_vals[c, s] = expectand(*arg_vals)

  return pushforward_vals
```

Regardless of whether the expectand samples were generated by `Stan` or if we had to derive them ourselves the Markov chain Monte Carlo estimation is the same. In particular we can estimate expectation values using either a single Markov chain or an entire ensemble of Markov chains.

```
# Estimate expectand expectation value from a single Markov chain.
# @param vals A one-dimensional array of sequential expectand values.
# @return The Markov chain Monte Carlo estimate, its estimated standard
#         error, and empirical effective sample size.
def mcmc_est(vals):
  """Estimate expectand expectation value from a Markov chain"""
  S = len(vals)
  if S == 1:
    return [vals[0], 0, math.nan]

  summary = welford_summary(vals)

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

  tau_hat = compute_tau_hat(vals)
  ess_hat = S / tau_hat
  return [summary[0], math.sqrt(summary[1] / ess_hat), ess_hat]
```

```
# Estimate expectand expectation value from a Markov chain ensemble.
# @param expectand_vals A two-dimensional array of expectand values with
#                       the first dimension indexing the Markov chains
#                       and the second dimension indexing the sequential
#                       states within each Markov chain.
# @return The ensemble Markov chain Monte Carlo estimate, its estimated
#         standard error, and empirical effective sample size.
def ensemble_mcmc_est(expectand_vals):
  """Estimate expectand expectation value from a collection of
     Markov chains"""
  validate_array(expectand_vals, 'expectand_vals')
```

```
  C = expectand_vals.shape[0]
  chain_ests = [ mcmc_est(expectand_vals[c,:]) for c in range(C) ]

  # 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]
```

We can also use realized Markov chains to estimate quantiles of the pushforward distribution along an expectand. Within a single Markov chain ordering the expectand values allow us to efficiently search for the value x_q whose corresponding interval probability first exceeds the defining quantile probability p,

$$\begin{aligned} p &< \pi(\{-\infty, x_q]) \\ &= \mathbb{E}_{\pi} \left[I_{\{-\infty, x_q]} \right] \\ &\approx \frac{1}{N} \sum_{n=1}^{N} I_{\{-\infty, x_q]}(\tilde{x}_n). \end{aligned}$$

The empirical quantiles within each Markov chain can then be averaged together to provide an ensemble estimator.

In theory the empirical standard deviation of the individual Markov chain estimates consistently estimates the estimator error, but the estimation is unreliable without many Markov chains. Consequently it is not reported here.

```
# Estimate expectand pushforward quantiles from a Markov chain ensemble.
# @param expectand_vals A two-dimensional array of expectand values with
#                       the first dimension indexing the Markov chains
#                       and the second dimension indexing the sequential
#                       states within each Markov chain.
# @param probs An array of quantile percentages in [0, 100].
# @return The ensemble Markov chain Monte Carlo quantile estimate.
def ensemble_mcmc_quantile_est(expectand_vals, probs):
  # Validate inputs
  validate_array(expectand_vals, 'expectand_vals')

  if not isinstance(probs, list):
    raise TypeError(('Input variable `probs` is not a list.'))

  # Estimate and return quantile
  q = numpy.zeros(len(probs))

  C = expectand_vals.shape[0]
  for c in range(C):
    q += numpy.percentile(expectand_vals[c,:], probs) / C

  return q
```

Finally we can also visualize the entire pushforward distribution by estimating the target probabilities in histogram bins.

```
# Visualize pushforward distribution of a given expectand as a
# histogram, using Markov chain Monte Carlo estimators to estimate the
# output bin probabilities.  Bin probability estimator error is shown
# in gray.
# @ax Matplotlib axis object
# @param expectand_vals A two-dimensional array of expectand values with
#                       the first dimension indexing the Markov chains
#                       and the second dimension indexing the sequential
#                       states within each Markov chain.
# @param B The number of histogram bins
# @param display_name Expectand name
# @param flim Optional histogram range
```

```
# @param ylim Optional y-axis range; ignored if add is TRUE
# @param color Color for plotting weighted bin probabilities; defaults
#              to dark.
# @param border Color for plotting estimator error; defaults to gray
# @param border_opacity Opacity for plotting estimator error; defaults
#                       to 1.
# @param add Configure plot to overlay over existing plot; defaults to
#            FALSE
# @param title Optional plot title
# @param baseline Optional baseline value for visual comparison
# @param baseline_color Color for plotting baseline value; defaults to
#                       "black"
def plot_expectand_pushforward(ax, expectand_vals, B, display_name="f",
                               flim=None, ylim=None,
                               color=dark, border="#DDDDDD",
                               border_opacity=1,
                               add=False, title=None,
                               baseline=None, baseline_color="black"):
  """Plot pushforward histogram of a given expectand using Markov chain
     Monte Carlo estimators to estimate the output bin probabilities"""
  validate_array(expectand_vals, 'expectand_vals')

  if flim is None:
    # Automatically adjust histogram binning to range of outputs
    min_f = min(expectand_vals.flatten())
    max_f = max(expectand_vals.flatten())
    delta = (max_f - min_f) / B

    # Add bounding bins
    B = B + 2
    min_f = min_f - delta
    max_f = max_f + delta
    flim = [min_f, max_f]

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

    bins = numpy.arange(min_f, max_f + delta, delta)
```

```
# Check sample containment
S = expectand_vals.size

S_low = sum(expectand_vals.flatten() < min_f)
if S_low == 1:
  print(f'{S_low} value ({S_low / S:.2%})'
        ' fell below the histogram binning.')
elif S_low > 1:
  print(f'{S_low} values ({S_low / S:.2%})'
        ' fell below the histogram binning.')

S_high = sum(max_f < expectand_vals.flatten())
if S_high == 1:
  print(f'{S_high} value ({S_high / S:.2%})'
        ' fell above the histogram binning.')
elif S_high > 1:
  print(f'{S_high} values ({S_high / S:.2%})'
        ' fell above the histogram binning.')

# Compute bin heights
mean_p = [0] * B
delta_p = [0] * B

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_vals = eval_uni_expectand_pushforward(expectand_vals,
                                                  bin_indicator)
  est = ensemble_mcmc_est(indicator_vals)

  # 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 ]

  if add:
    ax.fill_between(xs, lower_inter, upper_inter,
                    color=border, facecolor=border,
                    alpha=border_opacity)
    ax.plot(xs, [ mean_p[idx] for idx in idxs ],
            color=color, linewidth=2)
  else:
    if ylim is None:
      ylim = [ 0, 1.05 * max(upper_inter) ]

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

    if title is not None:
      ax.set_title(title)
    ax.set_xlim(flim)
    ax.set_xlabel(display_name)
    ax.set_ylim(ylim)
    ax.set_ylabel("Estimated Bin\nProbabilities / Bin Width")
    ax.get_yaxis().set_visible(False)
    ax.spines["top"].set_visible(False)
    ax.spines["left"].set_visible(False)
    ax.spines["right"].set_visible(False)

  if baseline is not None:
    ax.axvline(x=baseline, linewidth=4, color="white")
    ax.axvline(x=baseline, linewidth=2, color=baseline_color)
```

5 Demonstration

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

First we setup our local `Python` environment.

```
import matplotlib
import matplotlib.pyplot as plot
plot.show()
plot.rcParams['figure.figsize'] = [6, 4]
plot.rcParams['figure.dpi'] = 100
plot.rcParams['font.family'] = "Serif"

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

import math
import numpy

# Needed to run through a jupyter kernel
import nest_asyncio
nest_asyncio.apply()

import stan
```

Next we source all of these diagnostics into a local namespace to avoid any conflicts with other functions.

```
import mcmc_analysis_tools_pystan3 as util
```

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

```
with open('stan_programs/simu_logistic_reg.stan', 'r') as file:
  stan_program = file.read()
model = stan.build(stan_program, random_seed=4838282)
simu = model.fixed_param(num_chains=1, num_samples=1)

samples = util.extract_expectand_vals(simu)

y = [ v[0][0].astype(numpy.int64) for k, v in samples.items() if 'y' in k ]

N = 1000
M = 3
```

```
X = numpy.zeros((N, M))
for n in range(N):
  for m in range(M):
    X[n, m] = samples[f'X[{n + 1},{m + 1}]'][0, 0]

data = {'M': M, 'N': N, 'x0': [-1, 0, 1], 'X': X, 'y': y}
```

```
Building...
```

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.

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)
```

```
with open('stan_programs/bernoulli_linear.stan', 'r') as file:
  stan_program = file.read()
model = stan.build(stan_program, random_seed=8438338, data=data)
fit = model.sample(num_samples=1024, init=interval_inits)
```

```
Building...
```

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.

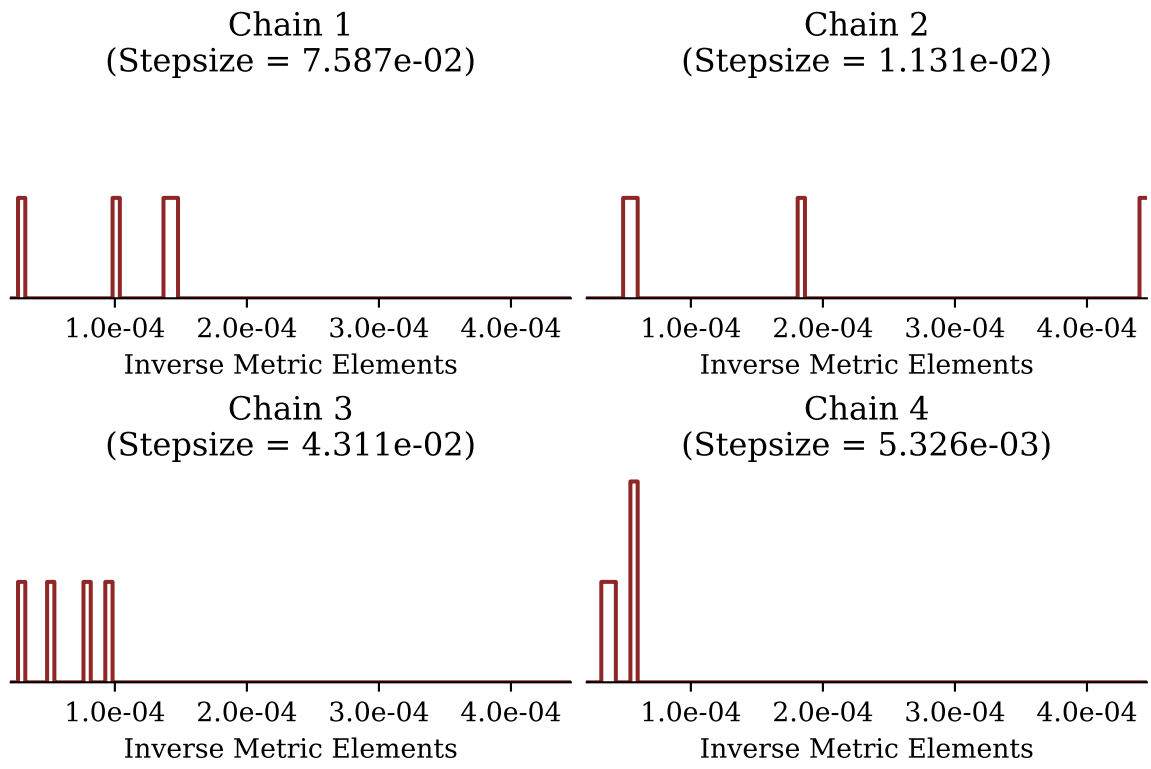
```
diagnostics = util.extract_hmc_diagnostics(fit)
util.check_all_hmc_diagnostics(diagnostics)
```

```
  Chain 1: 1016 of 1024 transitions (99.22%) diverged.
  Chain 1: Average proxy acceptance statistic (0.566) is smaller than
90% of the target (0.801).

  Chain 2: 1015 of 1024 transitions (99.12%) diverged.

  Chain 3: 1022 of 1024 transitions (99.80%) diverged.

  Chain 4: 1016 of 1024 transitions (99.22%) diverged.

Divergent Hamiltonian transitions result from unstable numerical
trajectories.  These instabilities are often due to degenerate target
geometry, especially "pinches".  If there are only a small number of
divergences then running with adept_delta larger than 0.801 may reduce
the instabilities at the cost of more expensive Hamiltonian transitions.

A small average proxy acceptance statistic indicates that the adaptation
of the numerical integrator step size failed to converge.  This is often
due to discontinuous or imprecise 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.

```
util.plot_inv_metric(fit, 75)
```



The problematic third Markov chain also exhibits the least 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. Inverse metric elements that cannot adapt to each parameter can frustrate numerical integration which can then frustrate the integrator step size adaptation.

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.

```
util.display_stepsizes(diagnostics)
```

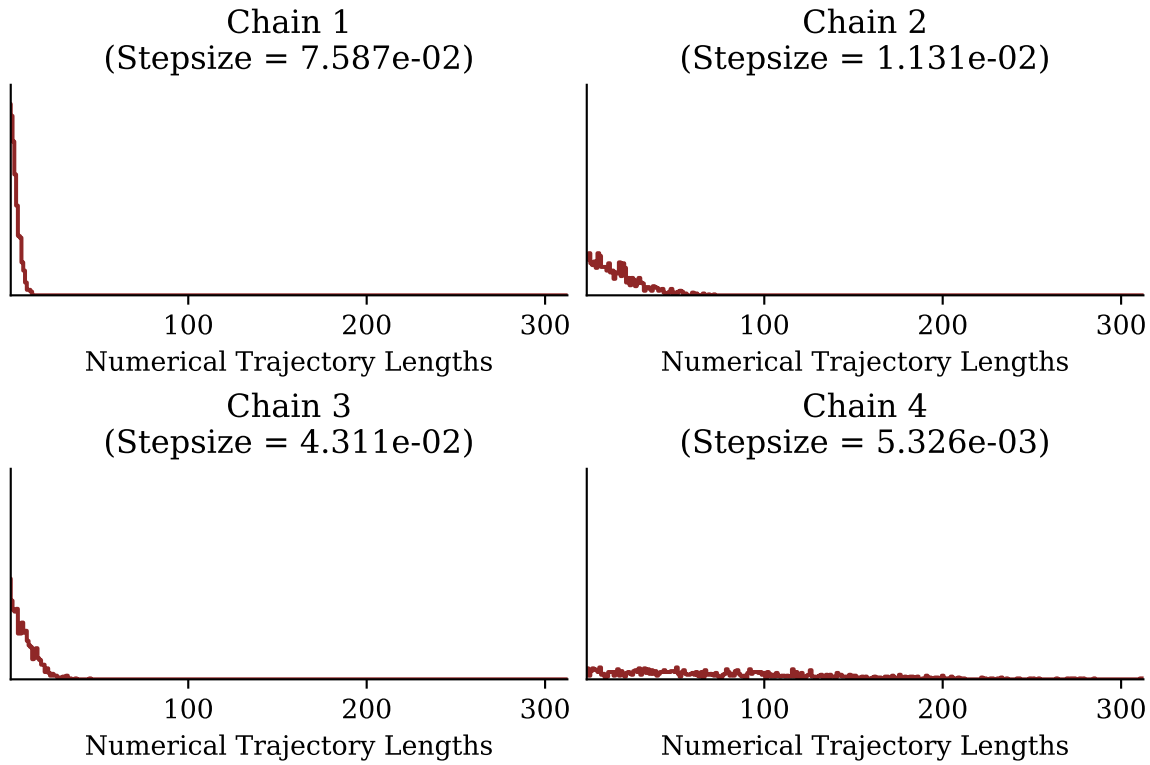
```
Chain 1: Integrator Step Size = 7.59e-02
Chain 2: Integrator Step Size = 1.13e-02
Chain 3: Integrator Step Size = 4.31e-02
Chain 4: Integrator Step Size = 5.33e-03
```

```
util.display_ave_accept_proxy(diagnostics)
```

```
Chain 1: Average proxy acceptance statistic = 0.566
Chain 2: Average proxy acceptance statistic = 0.856
Chain 3: Average proxy acceptance statistic = 0.738
Chain 4: Average proxy acceptance statistic = 0.957
```

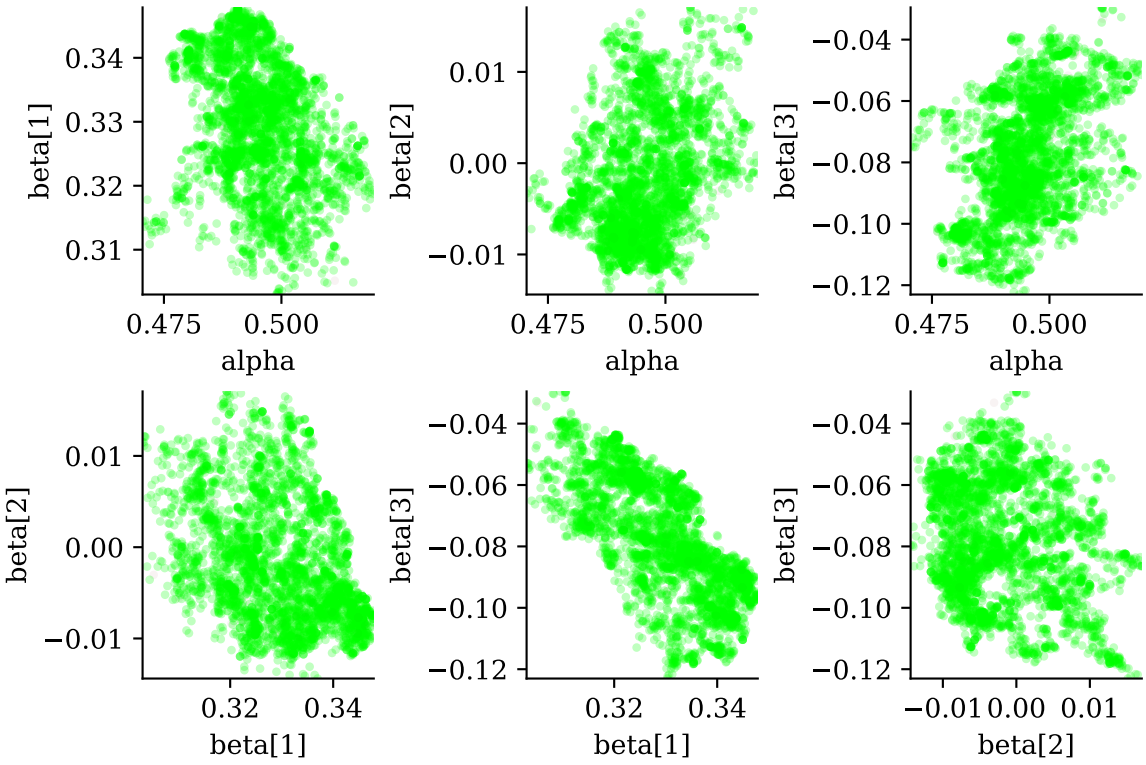
The different inverse metric results in different Hamiltonian dynamics. In this case the dynamics driving the third Markov chain are not able to explore as far as those in the other chains.

```
util.plot_num_leapfrogs_by_chain(diagnostics)
```



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

```
samples = util.extract_expectand_vals(fit)

names = ['alpha']
names += [ f'beta[{m + 1}]' for m in range(data['M']) ]
util.plot_div_pairs(names, names, samples, diagnostics)
```

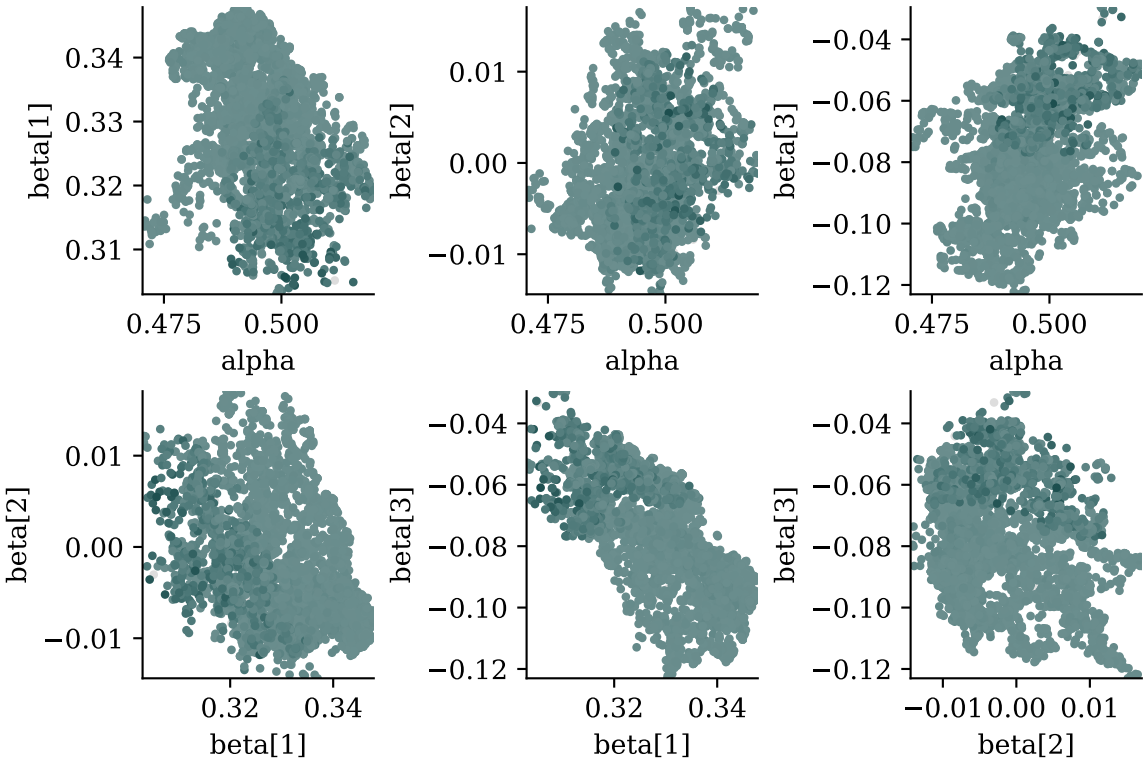


We can also color the divergent transitions by their numerical trajectory lengths. On average transitions from shorter numerical trajectories should be closer to the problematic behavior than transitions from longer numerical trajectories. Because there are so many divergent transitions here the point colors overlap and it's hard to make too much out, but there *may* be signs of a problematic boundary.

For example plot of `beta[2]` against `beta[1]` is not inconsistent with a boundary defined by

$$\beta_1 + \beta_2 = \text{constant}.$$

```
util.plot_div_pairs(names, names, samples, diagnostics, plot_mode=1)
```



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.

```
util.summarize_expectand_diagnostics(samples)
```

```
The expectands alpha, beta[1], beta[2], beta[3], p[1000], p[100],
p[101], p[102], p[103], p[104], p[105], p[106], p[107], p[108], p[109],
p[10], p[110], p[111], p[112], p[113], p[114], p[115], p[116], p[117],
p[118], p[119], p[11], p[120], p[121], p[122], p[123], p[124], p[125],
p[126], p[127], p[128], p[129], p[12], p[130], p[131], p[132], p[133],
p[134], p[135], p[136], p[137], p[138], p[139], p[13], p[140], p[141],
p[142], p[143], p[144], p[145], p[146], p[147], p[148], p[149], p[14],
p[150], p[151], p[152], p[153], p[154], p[155], p[156], p[157], p[158],
p[159], p[15], p[160], p[161], p[162], p[163], p[164], p[165], p[166],
p[167], p[168], p[169], p[16], p[170], p[171], p[172], p[173], p[174],
p[175], p[176], p[177], p[178], p[179], p[17], p[180], p[181], p[182],
```

p[183], p[184], p[185], p[186], p[187], p[188], p[189], p[18], p[190],
p[191], p[192], p[193], p[194], p[195], p[196], p[197], p[198], p[199],
p[19], p[1], p[200], p[201], p[202], p[203], p[204], p[205], p[206],
p[207], p[208], p[209], p[20], p[210], p[211], p[212], p[213], p[214],
p[215], p[216], p[217], p[218], p[219], p[21], p[220], p[221], p[222],
p[223], p[224], p[225], p[226], p[227], p[228], p[229], p[22], p[230],
p[231], p[232], p[233], p[234], p[235], p[236], p[237], p[238], p[239],
p[23], p[240], p[241], p[242], p[243], p[244], p[245], p[246], p[247],
p[248], p[249], p[24], p[250], p[251], p[252], p[253], p[254], p[255],
p[256], p[257], p[258], p[259], p[25], p[260], p[261], p[262], p[263],
p[264], p[265], p[266], p[267], p[268], p[269], p[26], p[270], p[271],
p[272], p[273], p[274], p[275], p[276], p[277], p[278], p[279], p[27],
p[280], p[281], p[282], p[283], p[284], p[285], p[286], p[287], p[288],
p[289], p[28], p[290], p[291], p[292], p[293], p[294], p[295], p[296],
p[297], p[298], p[299], p[29], p[2], p[300], p[301], p[302], p[303],
p[304], p[305], p[306], p[307], p[308], p[309], p[30], p[310], p[311],
p[312], p[313], p[314], p[315], p[316], p[317], p[318], p[319], p[31],
p[320], p[321], p[322], p[323], p[324], p[325], p[326], p[327], p[328],
p[329], p[32], p[330], p[331], p[332], p[333], p[334], p[335], p[336],
p[337], p[338], p[339], p[33], p[340], p[341], p[342], p[343], p[344],
p[345], p[346], p[347], p[348], p[349], p[34], p[350], p[351], p[352],
p[353], p[354], p[355], p[356], p[357], p[358], p[359], p[35], p[360],
p[361], p[362], p[363], p[364], p[365], p[366], p[367], p[368], p[369],
p[36], p[370], p[371], p[372], p[373], p[374], p[375], p[376], p[377],
p[378], p[379], p[37], p[380], p[381], p[382], p[383], p[384], p[385],
p[386], p[387], p[388], p[389], p[38], p[390], p[391], p[392], p[393],
p[394], p[395], p[396], p[397], p[398], p[399], p[39], p[3], p[400],
p[401], p[402], p[403], p[404], p[405], p[406], p[407], p[408], p[409],
p[40], p[410], p[411], p[412], p[413], p[414], p[415], p[416], p[417],
p[418], p[419], p[41], p[420], p[421], p[422], p[423], p[424], p[425],
p[426], p[427], p[428], p[429], p[42], p[430], p[431], p[432], p[433],
p[434], p[435], p[436], p[437], p[438], p[439], p[43], p[440], p[441],
p[442], p[443], p[444], p[445], p[446], p[447], p[448], p[449], p[44],
p[450], p[451], p[452], p[453], p[454], p[455], p[456], p[457], p[458],
p[459], p[45], p[460], p[461], p[462], p[463], p[464], p[465], p[466],
p[467], p[468], p[469], p[46], p[470], p[471], p[472], p[473], p[474],
p[475], p[476], p[477], p[478], p[479], p[47], p[480], p[481], p[482],
p[483], p[484], p[485], p[486], p[487], p[488], p[489], p[48], p[490],
p[491], p[492], p[493], p[494], p[495], p[496], p[497], p[498], p[499],
p[49], p[4], p[500], p[501], p[502], p[503], p[504], p[505], p[506],
p[507], p[508], p[509], p[50], p[510], p[511], p[512], p[513], p[514],
p[515], p[516], p[517], p[518], p[519], p[51], p[520], p[521], p[522],
p[523], p[524], p[525], p[526], p[527], p[528], p[529], p[52], p[530],

p[531], p[532], p[533], p[534], p[535], p[536], p[537], p[538], p[539],
p[53], p[540], p[541], p[542], p[543], p[544], p[545], p[546], p[547],
p[548], p[549], p[54], p[550], p[551], p[552], p[553], p[554], p[555],
p[556], p[557], p[558], p[559], p[55], p[560], p[561], p[562], p[563],
p[564], p[565], p[566], p[567], p[568], p[569], p[56], p[570], p[571],
p[572], p[573], p[574], p[575], p[576], p[577], p[578], p[579], p[57],
p[580], p[581], p[582], p[583], p[584], p[585], p[586], p[587], p[588],
p[589], p[58], p[590], p[591], p[592], p[593], p[594], p[595], p[596],
p[597], p[598], p[599], p[59], p[5], p[600], p[601], p[602], p[603],
p[604], p[605], p[606], p[607], p[608], p[609], p[60], p[610], p[611],
p[612], p[613], p[614], p[615], p[616], p[617], p[618], p[619], p[61],
p[620], p[621], p[622], p[623], p[624], p[625], p[626], p[627], p[628],
p[629], p[62], p[630], p[631], p[632], p[633], p[634], p[635], p[636],
p[637], p[638], p[639], p[63], p[640], p[641], p[642], p[643], p[644],
p[645], p[646], p[647], p[648], p[649], p[64], p[650], p[651], p[652],
p[653], p[654], p[655], p[656], p[657], p[658], p[659], p[65], p[660],
p[661], p[662], p[663], p[664], p[665], p[666], p[667], p[668], p[669],
p[66], p[670], p[671], p[672], p[673], p[674], p[675], p[676], p[677],
p[678], p[679], p[67], p[680], p[681], p[682], p[683], p[684], p[685],
p[686], p[687], p[688], p[689], p[68], p[690], p[691], p[692], p[693],
p[694], p[695], p[696], p[697], p[698], p[699], p[69], p[6], p[700],
p[701], p[702], p[703], p[704], p[705], p[706], p[707], p[708], p[709],
p[70], p[710], p[711], p[712], p[713], p[714], p[715], p[716], p[717],
p[718], p[719], p[71], p[720], p[721], p[722], p[723], p[724], p[725],
p[726], p[727], p[728], p[729], p[72], p[730], p[731], p[732], p[733],
p[734], p[735], p[736], p[737], p[738], p[739], p[73], p[740], p[741],
p[742], p[743], p[744], p[745], p[746], p[747], p[748], p[749], p[74],
p[750], p[751], p[752], p[753], p[754], p[755], p[756], p[757], p[758],
p[759], p[75], p[760], p[761], p[762], p[763], p[764], p[765], p[766],
p[767], p[768], p[769], p[76], p[770], p[771], p[772], p[773], p[774],
p[775], p[776], p[777], p[778], p[779], p[77], p[780], p[781], p[782],
p[783], p[784], p[785], p[786], p[787], p[788], p[789], p[78], p[790],
p[791], p[792], p[793], p[794], p[795], p[796], p[797], p[798], p[799],
p[79], p[7], p[800], p[801], p[802], p[803], p[804], p[805], p[806],
p[807], p[808], p[809], p[80], p[810], p[811], p[812], p[813], p[814],
p[815], p[816], p[817], p[818], p[819], p[81], p[820], p[821], p[822],
p[823], p[824], p[825], p[826], p[827], p[828], p[829], p[82], p[830],
p[831], p[832], p[833], p[834], p[835], p[836], p[837], p[838], p[839],
p[83], p[840], p[841], p[842], p[843], p[844], p[845], p[846], p[847],
p[848], p[849], p[84], p[850], p[851], p[852], p[853], p[854], p[855],
p[856], p[857], p[858], p[859], p[85], p[860], p[861], p[862], p[863],
p[864], p[865], p[866], p[867], p[868], p[869], p[86], p[870], p[871],
p[872], p[873], p[874], p[875], p[876], p[877], p[878], p[879], p[87],

p[880], p[881], p[882], p[883], p[884], p[885], p[886], p[887], p[888],
p[889], p[88], p[890], p[891], p[892], p[893], p[894], p[895], p[896],
p[897], p[898], p[899], p[89], p[8], p[900], p[901], p[902], p[903],
p[904], p[905], p[906], p[907], p[908], p[909], p[90], p[910], p[911],
p[912], p[913], p[914], p[915], p[916], p[917], p[918], p[919], p[91],
p[920], p[921], p[922], p[923], p[924], p[925], p[926], p[927], p[928],
p[929], p[92], p[930], p[931], p[932], p[933], p[934], p[935], p[936],
p[937], p[938], p[939], p[93], p[940], p[941], p[942], p[943], p[944],
p[945], p[946], p[947], p[948], p[949], p[94], p[950], p[951], p[952],
p[953], p[954], p[955], p[956], p[957], p[958], p[959], p[95], p[960],
p[961], p[962], p[963], p[964], p[965], p[966], p[967], p[968], p[969],
p[96], p[970], p[971], p[972], p[973], p[974], p[975], p[976], p[977],
p[978], p[979], p[97], p[980], p[981], p[982], p[983], p[984], p[985],
p[986], p[987], p[988], p[989], p[98], p[990], p[991], p[992], p[993],
p[994], p[995], p[996], p[997], p[998], p[999], p[99], p[9],
y_pred[100], y_pred[101], y_pred[106], y_pred[108], y_pred[10],
y_pred[114], y_pred[116], y_pred[121], y_pred[122], y_pred[123],
y_pred[124], y_pred[126], y_pred[127], y_pred[130], y_pred[131],
y_pred[132], y_pred[133], y_pred[134], y_pred[137], y_pred[138],
y_pred[13], y_pred[144], y_pred[149], y_pred[151], y_pred[153],
y_pred[154], y_pred[155], y_pred[156], y_pred[157], y_pred[15],
y_pred[160], y_pred[161], y_pred[163], y_pred[164], y_pred[165],
y_pred[166], y_pred[167], y_pred[16], y_pred[171], y_pred[173],
y_pred[176], y_pred[177], y_pred[179], y_pred[17], y_pred[181],
y_pred[182], y_pred[183], y_pred[184], y_pred[186], y_pred[187],
y_pred[189], y_pred[18], y_pred[191], y_pred[194], y_pred[195],
y_pred[197], y_pred[1], y_pred[200], y_pred[201], y_pred[202],
y_pred[204], y_pred[205], y_pred[206], y_pred[207], y_pred[210],
y_pred[211], y_pred[214], y_pred[216], y_pred[217], y_pred[219],
y_pred[225], y_pred[226], y_pred[227], y_pred[228], y_pred[233],
y_pred[236], y_pred[239], y_pred[240], y_pred[241], y_pred[243],
y_pred[244], y_pred[246], y_pred[249], y_pred[24], y_pred[250],
y_pred[251], y_pred[252], y_pred[253], y_pred[255], y_pred[256],
y_pred[257], y_pred[258], y_pred[260], y_pred[261], y_pred[265],
y_pred[266], y_pred[267], y_pred[268], y_pred[26], y_pred[270],
y_pred[272], y_pred[27], y_pred[281], y_pred[282], y_pred[283],
y_pred[284], y_pred[286], y_pred[287], y_pred[289], y_pred[290],
y_pred[291], y_pred[294], y_pred[296], y_pred[297], y_pred[298],
y_pred[29], y_pred[2], y_pred[300], y_pred[301], y_pred[303],
y_pred[304], y_pred[308], y_pred[309], y_pred[30], y_pred[310],
y_pred[315], y_pred[316], y_pred[318], y_pred[319], y_pred[31],
y_pred[324], y_pred[325], y_pred[326], y_pred[328], y_pred[32],
y_pred[330], y_pred[332], y_pred[333], y_pred[335], y_pred[336],

```
y_pred[338], y_pred[339], y_pred[33], y_pred[341], y_pred[342],
y_pred[344], y_pred[345], y_pred[347], y_pred[348], y_pred[349],
y_pred[352], y_pred[353], y_pred[356], y_pred[359], y_pred[35],
y_pred[364], y_pred[365], y_pred[366], y_pred[367], y_pred[371],
y_pred[372], y_pred[373], y_pred[376], y_pred[379], y_pred[381],
y_pred[382], y_pred[384], y_pred[386], y_pred[387], y_pred[388],
y_pred[391], y_pred[392], y_pred[393], y_pred[395], y_pred[397],
y_pred[398], y_pred[399], y_pred[3], y_pred[400], y_pred[401],
y_pred[403], y_pred[404], y_pred[405], y_pred[40], y_pred[410],
y_pred[413], y_pred[414], y_pred[415], y_pred[417], y_pred[418],
y_pred[41], y_pred[422], y_pred[423], y_pred[427], y_pred[430],
y_pred[435], y_pred[436], y_pred[437], y_pred[439], y_pred[43],
y_pred[440], y_pred[441], y_pred[445], y_pred[447], y_pred[449],
y_pred[451], y_pred[452], y_pred[454], y_pred[456], y_pred[457],
y_pred[459], y_pred[45], y_pred[460], y_pred[462], y_pred[463],
y_pred[464], y_pred[465], y_pred[468], y_pred[469], y_pred[473],
y_pred[474], y_pred[475], y_pred[476], y_pred[477], y_pred[479],
y_pred[47], y_pred[480], y_pred[481], y_pred[482], y_pred[484],
y_pred[485], y_pred[487], y_pred[488], y_pred[489], y_pred[490],
y_pred[491], y_pred[492], y_pred[494], y_pred[495], y_pred[496],
y_pred[498], y_pred[500], y_pred[503], y_pred[504], y_pred[506],
y_pred[508], y_pred[509], y_pred[50], y_pred[510], y_pred[511],
y_pred[513], y_pred[517], y_pred[518], y_pred[51], y_pred[521],
y_pred[522], y_pred[523], y_pred[524], y_pred[52], y_pred[530],
y_pred[531], y_pred[536], y_pred[537], y_pred[539], y_pred[53],
y_pred[540], y_pred[543], y_pred[544], y_pred[545], y_pred[547],
y_pred[549], y_pred[551], y_pred[553], y_pred[554], y_pred[555],
y_pred[559], y_pred[55], y_pred[562], y_pred[564], y_pred[565],
y_pred[56], y_pred[570], y_pred[571], y_pred[573], y_pred[575],
y_pred[576], y_pred[577], y_pred[578], y_pred[579], y_pred[57],
y_pred[584], y_pred[586], y_pred[590], y_pred[592], y_pred[593],
y_pred[595], y_pred[597], y_pred[598], y_pred[599], y_pred[59],
y_pred[601], y_pred[604], y_pred[606], y_pred[608], y_pred[609],
y_pred[60], y_pred[611], y_pred[612], y_pred[613], y_pred[614],
y_pred[617], y_pred[622], y_pred[624], y_pred[629], y_pred[62],
y_pred[630], y_pred[632], y_pred[634], y_pred[635], y_pred[636],
y_pred[637], y_pred[640], y_pred[646], y_pred[647], y_pred[648],
y_pred[64], y_pred[651], y_pred[657], y_pred[65], y_pred[661],
y_pred[668], y_pred[66], y_pred[670], y_pred[671], y_pred[672],
y_pred[675], y_pred[678], y_pred[679], y_pred[67], y_pred[680],
y_pred[684], y_pred[686], y_pred[689], y_pred[68], y_pred[690],
y_pred[691], y_pred[692], y_pred[694], y_pred[696], y_pred[698],
y_pred[699], y_pred[69], y_pred[700], y_pred[701], y_pred[703],
```

```
y_pred[704], y_pred[710], y_pred[713], y_pred[714], y_pred[715],
y_pred[716], y_pred[71], y_pred[722], y_pred[726], y_pred[728],
y_pred[729], y_pred[730], y_pred[733], y_pred[735], y_pred[736],
y_pred[737], y_pred[73], y_pred[742], y_pred[743], y_pred[744],
y_pred[745], y_pred[747], y_pred[74], y_pred[754], y_pred[755],
y_pred[757], y_pred[758], y_pred[760], y_pred[763], y_pred[764],
y_pred[767], y_pred[770], y_pred[771], y_pred[772], y_pred[775],
y_pred[776], y_pred[778], y_pred[779], y_pred[780], y_pred[781],
y_pred[784], y_pred[785], y_pred[792], y_pred[794], y_pred[798],
y_pred[799], y_pred[79], y_pred[7], y_pred[801], y_pred[805],
y_pred[806], y_pred[808], y_pred[80], y_pred[812], y_pred[814],
y_pred[815], y_pred[816], y_pred[817], y_pred[818], y_pred[81],
y_pred[821], y_pred[822], y_pred[823], y_pred[824], y_pred[825],
y_pred[826], y_pred[827], y_pred[828], y_pred[82], y_pred[830],
y_pred[832], y_pred[834], y_pred[836], y_pred[838], y_pred[840],
y_pred[843], y_pred[844], y_pred[84], y_pred[853], y_pred[855],
y_pred[857], y_pred[859], y_pred[85], y_pred[860], y_pred[862],
y_pred[863], y_pred[864], y_pred[867], y_pred[868], y_pred[869],
y_pred[870], y_pred[872], y_pred[874], y_pred[876], y_pred[882],
y_pred[886], y_pred[887], y_pred[889], y_pred[891], y_pred[892],
y_pred[894], y_pred[898], y_pred[899], y_pred[89], y_pred[8],
y_pred[900], y_pred[905], y_pred[90], y_pred[911], y_pred[912],
y_pred[915], y_pred[916], y_pred[917], y_pred[918], y_pred[919],
y_pred[921], y_pred[927], y_pred[928], y_pred[930], y_pred[933],
y_pred[934], y_pred[936], y_pred[937], y_pred[938], y_pred[941],
y_pred[942], y_pred[944], y_pred[947], y_pred[949], y_pred[94],
y_pred[952], y_pred[953], y_pred[954], y_pred[957], y_pred[958],
y_pred[960], y_pred[962], y_pred[963], y_pred[967], y_pred[969],
y_pred[970], y_pred[971], y_pred[972], y_pred[974], y_pred[975],
y_pred[976], y_pred[977], y_pred[979], y_pred[97], y_pred[981],
y_pred[982], y_pred[986], y_pred[987], y_pred[991], y_pred[992],
y_pred[994], y_pred[995], y_pred[996], y_pred[997], y_pred[99],
y_pred[9] triggered diagnostic warnings.

The expectands y_pred[100], y_pred[101], y_pred[106], y_pred[108],
y_pred[10], y_pred[114], y_pred[116], y_pred[121], y_pred[122],
y_pred[123], y_pred[124], y_pred[126], y_pred[127], y_pred[130],
y_pred[131], y_pred[132], y_pred[133], y_pred[134], y_pred[137],
y_pred[138], y_pred[13], y_pred[144], y_pred[149], y_pred[151],
y_pred[153], y_pred[154], y_pred[155], y_pred[156], y_pred[157],
y_pred[15], y_pred[160], y_pred[161], y_pred[163], y_pred[164],
y_pred[165], y_pred[166], y_pred[167], y_pred[16], y_pred[171],
y_pred[173], y_pred[176], y_pred[177], y_pred[179], y_pred[17],
```

```
y_pred[181], y_pred[182], y_pred[183], y_pred[184], y_pred[186],
y_pred[187], y_pred[189], y_pred[18], y_pred[191], y_pred[194],
y_pred[195], y_pred[197], y_pred[1], y_pred[200], y_pred[201],
y_pred[202], y_pred[204], y_pred[205], y_pred[206], y_pred[207],
y_pred[210], y_pred[211], y_pred[214], y_pred[216], y_pred[217],
y_pred[219], y_pred[225], y_pred[226], y_pred[227], y_pred[228],
y_pred[233], y_pred[236], y_pred[239], y_pred[240], y_pred[241],
y_pred[243], y_pred[244], y_pred[246], y_pred[249], y_pred[24],
y_pred[250], y_pred[251], y_pred[252], y_pred[253], y_pred[255],
y_pred[256], y_pred[257], y_pred[258], y_pred[260], y_pred[261],
y_pred[265], y_pred[266], y_pred[267], y_pred[268], y_pred[26],
y_pred[270], y_pred[272], y_pred[27], y_pred[281], y_pred[282],
y_pred[283], y_pred[284], y_pred[286], y_pred[287], y_pred[289],
y_pred[290], y_pred[291], y_pred[294], y_pred[296], y_pred[297],
y_pred[298], y_pred[29], y_pred[2], y_pred[300], y_pred[301],
y_pred[303], y_pred[304], y_pred[308], y_pred[309], y_pred[30],
y_pred[310], y_pred[315], y_pred[316], y_pred[318], y_pred[319],
y_pred[31], y_pred[324], y_pred[325], y_pred[326], y_pred[328],
y_pred[32], y_pred[330], y_pred[332], y_pred[333], y_pred[335],
y_pred[336], y_pred[338], y_pred[339], y_pred[33], y_pred[341],
y_pred[342], y_pred[344], y_pred[345], y_pred[347], y_pred[348],
y_pred[349], y_pred[352], y_pred[353], y_pred[356], y_pred[359],
y_pred[35], y_pred[364], y_pred[365], y_pred[366], y_pred[367],
y_pred[371], y_pred[372], y_pred[373], y_pred[376], y_pred[379],
y_pred[381], y_pred[382], y_pred[384], y_pred[386], y_pred[387],
y_pred[388], y_pred[391], y_pred[392], y_pred[393], y_pred[395],
y_pred[397], y_pred[398], y_pred[399], y_pred[3], y_pred[400],
y_pred[401], y_pred[403], y_pred[404], y_pred[405], y_pred[40],
y_pred[410], y_pred[413], y_pred[414], y_pred[415], y_pred[417],
y_pred[418], y_pred[41], y_pred[422], y_pred[423], y_pred[427],
y_pred[430], y_pred[435], y_pred[436], y_pred[437], y_pred[439],
y_pred[43], y_pred[440], y_pred[441], y_pred[445], y_pred[447],
y_pred[449], y_pred[451], y_pred[452], y_pred[454], y_pred[456],
y_pred[457], y_pred[459], y_pred[45], y_pred[460], y_pred[462],
y_pred[463], y_pred[464], y_pred[465], y_pred[468], y_pred[469],
y_pred[473], y_pred[474], y_pred[475], y_pred[476], y_pred[477],
y_pred[479], y_pred[47], y_pred[480], y_pred[481], y_pred[482],
y_pred[484], y_pred[485], y_pred[487], y_pred[488], y_pred[489],
y_pred[490], y_pred[491], y_pred[492], y_pred[494], y_pred[495],
y_pred[496], y_pred[498], y_pred[500], y_pred[503], y_pred[504],
y_pred[506], y_pred[508], y_pred[509], y_pred[50], y_pred[510],
y_pred[511], y_pred[513], y_pred[517], y_pred[518], y_pred[51],
y_pred[521], y_pred[522], y_pred[523], y_pred[524], y_pred[52],
```

```
y_pred[530], y_pred[531], y_pred[536], y_pred[537], y_pred[539],
y_pred[53], y_pred[540], y_pred[543], y_pred[544], y_pred[545],
y_pred[547], y_pred[549], y_pred[551], y_pred[553], y_pred[554],
y_pred[555], y_pred[559], y_pred[55], y_pred[562], y_pred[564],
y_pred[565], y_pred[56], y_pred[570], y_pred[571], y_pred[573],
y_pred[575], y_pred[576], y_pred[577], y_pred[578], y_pred[579],
y_pred[57], y_pred[584], y_pred[586], y_pred[590], y_pred[592],
y_pred[593], y_pred[595], y_pred[597], y_pred[598], y_pred[599],
y_pred[59], y_pred[601], y_pred[604], y_pred[606], y_pred[608],
y_pred[609], y_pred[60], y_pred[611], y_pred[612], y_pred[613],
y_pred[614], y_pred[617], y_pred[622], y_pred[624], y_pred[629],
y_pred[62], y_pred[630], y_pred[632], y_pred[634], y_pred[635],
y_pred[636], y_pred[637], y_pred[640], y_pred[646], y_pred[647],
y_pred[648], y_pred[64], y_pred[651], y_pred[657], y_pred[65],
y_pred[661], y_pred[668], y_pred[66], y_pred[670], y_pred[671],
y_pred[672], y_pred[675], y_pred[678], y_pred[679], y_pred[67],
y_pred[680], y_pred[684], y_pred[686], y_pred[689], y_pred[68],
y_pred[690], y_pred[691], y_pred[692], y_pred[694], y_pred[696],
y_pred[698], y_pred[699], y_pred[69], y_pred[700], y_pred[701],
y_pred[703], y_pred[704], y_pred[710], y_pred[713], y_pred[714],
y_pred[715], y_pred[716], y_pred[71], y_pred[722], y_pred[726],
y_pred[728], y_pred[729], y_pred[730], y_pred[733], y_pred[735],
y_pred[736], y_pred[737], y_pred[73], y_pred[742], y_pred[743],
y_pred[744], y_pred[745], y_pred[747], y_pred[74], y_pred[754],
y_pred[755], y_pred[757], y_pred[758], y_pred[760], y_pred[763],
y_pred[764], y_pred[767], y_pred[770], y_pred[771], y_pred[772],
y_pred[775], y_pred[776], y_pred[778], y_pred[779], y_pred[780],
y_pred[781], y_pred[784], y_pred[785], y_pred[792], y_pred[794],
y_pred[798], y_pred[799], y_pred[79], y_pred[7], y_pred[801],
y_pred[805], y_pred[806], y_pred[808], y_pred[80], y_pred[812],
y_pred[814], y_pred[815], y_pred[816], y_pred[817], y_pred[818],
y_pred[81], y_pred[821], y_pred[822], y_pred[823], y_pred[824],
y_pred[825], y_pred[826], y_pred[827], y_pred[828], y_pred[82],
y_pred[830], y_pred[832], y_pred[834], y_pred[836], y_pred[838],
y_pred[840], y_pred[843], y_pred[844], y_pred[84], y_pred[853],
y_pred[855], y_pred[857], y_pred[859], y_pred[85], y_pred[860],
y_pred[862], y_pred[863], y_pred[864], y_pred[867], y_pred[868],
y_pred[869], y_pred[870], y_pred[872], y_pred[874], y_pred[876],
y_pred[882], y_pred[886], y_pred[887], y_pred[889], y_pred[891],
y_pred[892], y_pred[894], y_pred[898], y_pred[899], y_pred[89],
y_pred[8], y_pred[900], y_pred[905], y_pred[90], y_pred[911],
y_pred[912], y_pred[915], y_pred[916], y_pred[917], y_pred[918],
y_pred[919], y_pred[921], y_pred[927], y_pred[928], y_pred[930],
```

```
y_pred[933], y_pred[934], y_pred[936], y_pred[937], y_pred[938],
y_pred[941], y_pred[942], y_pred[944], y_pred[947], y_pred[949],
y_pred[94], y_pred[952], y_pred[953], y_pred[954], y_pred[957],
y_pred[958], y_pred[960], y_pred[962], y_pred[963], y_pred[967],
y_pred[969], y_pred[970], y_pred[971], y_pred[972], y_pred[974],
y_pred[975], y_pred[976], y_pred[977], y_pred[979], y_pred[97],
y_pred[981], y_pred[982], y_pred[986], y_pred[987], y_pred[991],
y_pred[992], y_pred[994], y_pred[995], y_pred[996], y_pred[997],
y_pred[99], y_pred[9] triggered tail hat{xi} warnings.
  Large tail hat{xi}s suggest that the expectand might not be
sufficiently integrable.

The expectands alpha, beta[1], beta[2], beta[3], p[1000], p[100],
p[101], p[102], p[103], p[105], p[106], p[107], p[108], p[109], p[10],
p[110], p[111], p[112], p[113], p[114], p[116], p[118], p[119], p[11],
p[120], p[121], p[122], p[123], p[124], p[125], p[126], p[127], p[128],
p[129], p[130], p[132], p[133], p[134], p[135], p[136], p[137], p[139],
p[13], p[140], p[141], p[142], p[143], p[144], p[145], p[146], p[147],
p[148], p[14], p[150], p[151], p[152], p[153], p[154], p[155], p[156],
p[157], p[158], p[159], p[160], p[161], p[162], p[163], p[164], p[165],
p[166], p[167], p[168], p[169], p[16], p[170], p[171], p[172], p[174],
p[175], p[176], p[177], p[179], p[17], p[180], p[181], p[182], p[184],
p[185], p[186], p[187], p[188], p[189], p[18], p[190], p[191], p[192],
p[193], p[195], p[196], p[197], p[198], p[199], p[19], p[1], p[200],
p[201], p[202], p[203], p[204], p[205], p[206], p[207], p[208], p[209],
p[20], p[210], p[211], p[212], p[213], p[214], p[215], p[216], p[217],
p[218], p[219], p[21], p[222], p[223], p[225], p[226], p[227], p[228],
p[229], p[22], p[230], p[231], p[232], p[234], p[235], p[236], p[237],
p[238], p[239], p[240], p[241], p[242], p[243], p[244], p[245], p[246],
p[248], p[249], p[24], p[250], p[251], p[252], p[254], p[255], p[256],
p[257], p[258], p[259], p[25], p[260], p[262], p[263], p[264], p[265],
p[266], p[267], p[268], p[269], p[26], p[270], p[271], p[272], p[273],
p[274], p[275], p[276], p[277], p[278], p[279], p[280], p[281], p[282],
p[283], p[285], p[286], p[287], p[288], p[28], p[290], p[294], p[295],
p[297], p[298], p[299], p[29], p[2], p[300], p[301], p[302], p[303],
p[304], p[305], p[306], p[307], p[309], p[30], p[310], p[311], p[312],
p[313], p[314], p[315], p[316], p[317], p[318], p[319], p[320], p[322],
p[323], p[324], p[325], p[326], p[327], p[328], p[329], p[32], p[331],
p[332], p[333], p[334], p[335], p[336], p[337], p[33], p[340], p[341],
p[342], p[343], p[344], p[345], p[346], p[347], p[348], p[349], p[34],
p[350], p[351], p[352], p[353], p[354], p[356], p[357], p[358], p[359],
p[35], p[360], p[361], p[363], p[366], p[367], p[368], p[369], p[370],
p[371], p[372], p[373], p[374], p[375], p[376], p[377], p[378], p[379],
```

p[37], p[380], p[381], p[382], p[383], p[384], p[385], p[386], p[387],
p[388], p[389], p[38], p[390], p[391], p[392], p[394], p[396], p[397],
p[398], p[399], p[39], p[3], p[400], p[401], p[402], p[403], p[404],
p[405], p[407], p[408], p[409], p[40], p[410], p[411], p[412], p[413],
p[414], p[415], p[416], p[417], p[418], p[419], p[41], p[420], p[421],
p[422], p[423], p[424], p[425], p[426], p[428], p[429], p[42], p[430],
p[431], p[432], p[433], p[434], p[435], p[436], p[437], p[438], p[439],
p[43], p[440], p[441], p[443], p[444], p[445], p[446], p[447], p[448],
p[449], p[44], p[450], p[451], p[452], p[453], p[454], p[455], p[456],
p[457], p[458], p[459], p[460], p[461], p[462], p[463], p[464], p[465],
p[466], p[467], p[468], p[469], p[46], p[470], p[471], p[472], p[473],
p[474], p[475], p[476], p[477], p[478], p[47], p[480], p[481], p[482],
p[483], p[484], p[485], p[487], p[48], p[490], p[491], p[492], p[493],
p[494], p[495], p[496], p[497], p[498], p[499], p[49], p[4], p[501],
p[502], p[503], p[504], p[505], p[506], p[507], p[508], p[509], p[50],
p[510], p[511], p[512], p[514], p[515], p[516], p[517], p[518], p[519],
p[51], p[520], p[522], p[524], p[525], p[526], p[527], p[528], p[529],
p[52], p[530], p[531], p[532], p[533], p[534], p[535], p[536], p[537],
p[538], p[539], p[53], p[540], p[541], p[542], p[543], p[544], p[545],
p[546], p[547], p[548], p[549], p[54], p[550], p[551], p[552], p[553],
p[554], p[555], p[556], p[557], p[558], p[559], p[55], p[560], p[561],
p[562], p[563], p[564], p[566], p[567], p[568], p[569], p[56], p[570],
p[571], p[572], p[573], p[574], p[575], p[576], p[578], p[57], p[580],
p[581], p[582], p[583], p[584], p[585], p[586], p[587], p[588], p[589],
p[58], p[590], p[591], p[592], p[593], p[594], p[596], p[597], p[599],
p[59], p[5], p[600], p[601], p[602], p[604], p[605], p[606], p[607],
p[608], p[609], p[60], p[610], p[612], p[613], p[615], p[616], p[617],
p[618], p[619], p[61], p[620], p[621], p[622], p[623], p[624], p[625],
p[626], p[627], p[628], p[629], p[630], p[631], p[632], p[633], p[634],
p[635], p[636], p[637], p[638], p[639], p[63], p[640], p[641], p[642],
p[643], p[644], p[645], p[648], p[649], p[650], p[651], p[652], p[653],
p[654], p[655], p[656], p[657], p[658], p[659], p[65], p[662], p[663],
p[664], p[665], p[666], p[667], p[668], p[669], p[66], p[670], p[671],
p[672], p[673], p[674], p[675], p[676], p[677], p[678], p[679], p[67],
p[680], p[681], p[682], p[683], p[684], p[685], p[687], p[688], p[689],
p[690], p[691], p[692], p[693], p[694], p[695], p[697], p[698], p[69],
p[6], p[700], p[701], p[702], p[703], p[704], p[705], p[706], p[707],
p[708], p[709], p[70], p[710], p[712], p[714], p[715], p[717], p[719],
p[71], p[720], p[721], p[723], p[724], p[725], p[726], p[727], p[728],
p[729], p[72], p[730], p[731], p[732], p[733], p[734], p[735], p[736],
p[737], p[738], p[739], p[73], p[740], p[741], p[742], p[743], p[744],
p[745], p[746], p[747], p[748], p[749], p[74], p[750], p[751], p[752],
p[753], p[754], p[755], p[756], p[757], p[758], p[759], p[75], p[760],

```
p[761], p[762], p[764], p[765], p[766], p[767], p[768], p[769], p[76],
p[770], p[771], p[772], p[773], p[774], p[775], p[776], p[777], p[778],
p[779], p[77], p[780], p[781], p[782], p[783], p[784], p[785], p[786],
p[787], p[788], p[789], p[78], p[790], p[791], p[792], p[793], p[794],
p[795], p[797], p[798], p[799], p[79], p[7], p[801], p[802], p[803],
p[804], p[805], p[806], p[807], p[808], p[809], p[80], p[810], p[811],
p[812], p[813], p[817], p[818], p[819], p[81], p[820], p[821], p[822],
p[823], p[824], p[825], p[827], p[828], p[829], p[831], p[832], p[833],
p[834], p[835], p[836], p[837], p[838], p[839], p[83], p[841], p[842],
p[843], p[844], p[845], p[846], p[847], p[848], p[849], p[850], p[851],
p[852], p[854], p[855], p[856], p[857], p[858], p[859], p[85], p[860],
p[861], p[862], p[864], p[865], p[866], p[867], p[868], p[86], p[870],
p[871], p[872], p[873], p[874], p[875], p[876], p[877], p[878], p[879],
p[87], p[880], p[881], p[882], p[883], p[884], p[885], p[886], p[887],
p[888], p[889], p[88], p[890], p[891], p[892], p[893], p[895], p[896],
p[897], p[899], p[89], p[8], p[900], p[901], p[902], p[904], p[905],
p[906], p[907], p[908], p[909], p[90], p[910], p[911], p[913], p[914],
p[915], p[916], p[917], p[918], p[919], p[91], p[920], p[921], p[922],
p[923], p[924], p[925], p[926], p[927], p[928], p[929], p[92], p[930],
p[931], p[932], p[933], p[934], p[935], p[937], p[938], p[939], p[93],
p[940], p[941], p[942], p[943], p[944], p[945], p[946], p[947], p[948],
p[94], p[950], p[951], p[952], p[953], p[954], p[955], p[956], p[957],
p[958], p[959], p[95], p[960], p[961], p[962], p[963], p[964], p[965],
p[966], p[968], p[969], p[96], p[970], p[971], p[972], p[973], p[974],
p[975], p[976], p[977], p[978], p[979], p[97], p[980], p[981], p[982],
p[983], p[984], p[985], p[986], p[987], p[989], p[98], p[991], p[992],
p[993], p[994], p[995], p[996], p[997], p[998], p[999], p[99], p[9]
triggered hat{R} warnings.
  Split Rhat larger than 1.1 suggests that at least one of the Markov
chains has not reached an equilibrium.

The expectands alpha, beta[1], beta[2], beta[3], p[1000], p[100],
p[101], p[102], p[103], p[104], p[105], p[106], p[107], p[108], p[109],
p[10], p[110], p[111], p[112], p[113], p[114], p[115], p[116], p[117],
p[118], p[119], p[11], p[120], p[121], p[122], p[123], p[124], p[125],
p[126], p[127], p[128], p[129], p[12], p[130], p[131], p[132], p[133],
p[134], p[135], p[136], p[137], p[138], p[139], p[13], p[140], p[141],
p[142], p[143], p[144], p[145], p[146], p[147], p[148], p[149], p[14],
p[150], p[151], p[152], p[153], p[154], p[155], p[156], p[157], p[158],
p[159], p[15], p[160], p[161], p[162], p[163], p[164], p[165], p[166],
p[167], p[168], p[169], p[16], p[170], p[171], p[172], p[173], p[174],
p[175], p[176], p[177], p[178], p[179], p[17], p[180], p[181], p[182],
p[183], p[184], p[185], p[186], p[187], p[188], p[189], p[18], p[190],
```

p[191], p[192], p[193], p[194], p[195], p[196], p[197], p[198], p[199],
p[19], p[1], p[200], p[201], p[202], p[203], p[204], p[205], p[206],
p[207], p[208], p[209], p[20], p[210], p[211], p[212], p[213], p[214],
p[215], p[216], p[217], p[218], p[219], p[21], p[220], p[221], p[222],
p[223], p[224], p[225], p[226], p[227], p[228], p[229], p[22], p[230],
p[231], p[232], p[233], p[234], p[235], p[236], p[237], p[238], p[239],
p[23], p[240], p[241], p[242], p[243], p[244], p[245], p[246], p[247],
p[248], p[249], p[24], p[250], p[251], p[252], p[253], p[254], p[255],
p[256], p[257], p[258], p[259], p[25], p[260], p[261], p[262], p[263],
p[264], p[265], p[266], p[267], p[268], p[269], p[26], p[270], p[271],
p[272], p[273], p[274], p[275], p[276], p[277], p[278], p[279], p[27],
p[280], p[281], p[282], p[283], p[284], p[285], p[286], p[287], p[288],
p[289], p[28], p[290], p[291], p[292], p[293], p[294], p[295], p[296],
p[297], p[298], p[299], p[29], p[2], p[300], p[301], p[302], p[303],
p[304], p[305], p[306], p[307], p[308], p[309], p[30], p[310], p[311],
p[312], p[313], p[314], p[315], p[316], p[317], p[318], p[319], p[31],
p[320], p[321], p[322], p[323], p[324], p[325], p[326], p[327], p[328],
p[329], p[32], p[330], p[331], p[332], p[333], p[334], p[335], p[336],
p[337], p[338], p[339], p[33], p[340], p[341], p[342], p[343], p[344],
p[345], p[346], p[347], p[348], p[349], p[34], p[350], p[351], p[352],
p[353], p[354], p[355], p[356], p[357], p[358], p[359], p[35], p[360],
p[361], p[362], p[363], p[364], p[365], p[366], p[367], p[368], p[369],
p[36], p[370], p[371], p[372], p[373], p[374], p[375], p[376], p[377],
p[378], p[379], p[37], p[380], p[381], p[382], p[383], p[384], p[385],
p[386], p[387], p[388], p[389], p[38], p[390], p[391], p[392], p[393],
p[394], p[395], p[396], p[397], p[398], p[399], p[39], p[3], p[400],
p[401], p[402], p[403], p[404], p[405], p[406], p[407], p[408], p[409],
p[40], p[410], p[411], p[412], p[413], p[414], p[415], p[416], p[417],
p[418], p[419], p[41], p[420], p[421], p[422], p[423], p[424], p[425],
p[426], p[427], p[428], p[429], p[42], p[430], p[431], p[432], p[433],
p[434], p[435], p[436], p[437], p[438], p[439], p[43], p[440], p[441],
p[442], p[443], p[444], p[445], p[446], p[447], p[448], p[449], p[44],
p[450], p[451], p[452], p[453], p[454], p[455], p[456], p[457], p[458],
p[459], p[45], p[460], p[461], p[462], p[463], p[464], p[465], p[466],
p[467], p[468], p[469], p[46], p[470], p[471], p[472], p[473], p[474],
p[475], p[476], p[477], p[478], p[479], p[47], p[480], p[481], p[482],
p[483], p[484], p[485], p[486], p[487], p[488], p[489], p[48], p[490],
p[491], p[492], p[493], p[494], p[495], p[496], p[497], p[498], p[499],
p[49], p[4], p[500], p[501], p[502], p[503], p[504], p[505], p[506],
p[507], p[508], p[509], p[50], p[510], p[511], p[512], p[513], p[514],
p[515], p[516], p[517], p[518], p[519], p[51], p[520], p[521], p[522],
p[523], p[524], p[525], p[526], p[527], p[528], p[529], p[52], p[530],
p[531], p[532], p[533], p[534], p[535], p[536], p[537], p[538], p[539],

p[53], p[540], p[541], p[542], p[543], p[544], p[545], p[546], p[547],
p[548], p[549], p[54], p[550], p[551], p[552], p[553], p[554], p[555],
p[556], p[557], p[558], p[559], p[55], p[560], p[561], p[562], p[563],
p[564], p[565], p[566], p[567], p[568], p[569], p[56], p[570], p[571],
p[572], p[573], p[574], p[575], p[576], p[577], p[578], p[579], p[57],
p[580], p[581], p[582], p[583], p[584], p[585], p[586], p[587], p[588],
p[589], p[58], p[590], p[591], p[592], p[593], p[594], p[595], p[596],
p[597], p[598], p[599], p[59], p[5], p[600], p[601], p[602], p[603],
p[604], p[605], p[606], p[607], p[608], p[609], p[60], p[610], p[611],
p[612], p[613], p[614], p[615], p[616], p[617], p[618], p[619], p[61],
p[620], p[621], p[622], p[623], p[624], p[625], p[626], p[627], p[628],
p[629], p[62], p[630], p[631], p[632], p[633], p[634], p[635], p[636],
p[637], p[638], p[639], p[63], p[640], p[641], p[642], p[643], p[644],
p[645], p[646], p[647], p[648], p[649], p[64], p[650], p[651], p[652],
p[653], p[654], p[655], p[656], p[657], p[658], p[659], p[65], p[660],
p[661], p[662], p[663], p[664], p[665], p[666], p[667], p[668], p[669],
p[66], p[670], p[671], p[672], p[673], p[674], p[675], p[676], p[677],
p[678], p[679], p[67], p[680], p[681], p[682], p[683], p[684], p[685],
p[686], p[687], p[688], p[689], p[68], p[690], p[691], p[692], p[693],
p[694], p[695], p[696], p[697], p[698], p[699], p[69], p[6], p[700],
p[701], p[702], p[703], p[704], p[705], p[706], p[707], p[708], p[709],
p[70], p[710], p[711], p[712], p[713], p[714], p[715], p[716], p[717],
p[718], p[719], p[71], p[720], p[721], p[722], p[723], p[724], p[725],
p[726], p[727], p[728], p[729], p[72], p[730], p[731], p[732], p[733],
p[734], p[735], p[736], p[737], p[738], p[739], p[73], p[740], p[741],
p[742], p[743], p[744], p[745], p[746], p[747], p[748], p[749], p[74],
p[750], p[751], p[752], p[753], p[754], p[755], p[756], p[757], p[758],
p[759], p[75], p[760], p[761], p[762], p[763], p[764], p[765], p[766],
p[767], p[768], p[769], p[76], p[770], p[771], p[772], p[773], p[774],
p[775], p[776], p[777], p[778], p[779], p[77], p[780], p[781], p[782],
p[783], p[784], p[785], p[786], p[787], p[788], p[789], p[78], p[790],
p[791], p[792], p[793], p[794], p[795], p[796], p[797], p[798], p[799],
p[79], p[7], p[800], p[801], p[802], p[803], p[804], p[805], p[806],
p[807], p[808], p[809], p[80], p[810], p[811], p[812], p[813], p[814],
p[815], p[816], p[817], p[818], p[819], p[81], p[820], p[821], p[822],
p[823], p[824], p[825], p[826], p[827], p[828], p[829], p[82], p[830],
p[831], p[832], p[833], p[834], p[835], p[836], p[837], p[838], p[839],
p[83], p[840], p[841], p[842], p[843], p[844], p[845], p[846], p[847],
p[848], p[849], p[84], p[850], p[851], p[852], p[853], p[854], p[855],
p[856], p[857], p[858], p[859], p[85], p[860], p[861], p[862], p[863],
p[864], p[865], p[866], p[867], p[868], p[869], p[86], p[870], p[871],
p[872], p[873], p[874], p[875], p[876], p[877], p[878], p[879], p[87],
p[880], p[881], p[882], p[883], p[884], p[885], p[886], p[887], p[888],

```
p[889], p[88], p[890], p[891], p[892], p[893], p[894], p[895], p[896],
p[897], p[898], p[899], p[89], p[8], p[900], p[901], p[902], p[903],
p[904], p[905], p[906], p[907], p[908], p[909], p[90], p[910], p[911],
p[912], p[913], p[914], p[915], p[916], p[917], p[918], p[919], p[91],
p[920], p[921], p[922], p[923], p[924], p[925], p[926], p[927], p[928],
p[929], p[92], p[930], p[931], p[932], p[933], p[934], p[935], p[936],
p[937], p[938], p[939], p[93], p[940], p[941], p[942], p[943], p[944],
p[945], p[946], p[947], p[948], p[949], p[94], p[950], p[951], p[952],
p[953], p[954], p[955], p[956], p[957], p[958], p[959], p[95], p[960],
p[961], p[962], p[963], p[964], p[965], p[966], p[967], p[968], p[969],
p[96], p[970], p[971], p[972], p[973], p[974], p[975], p[976], p[977],
p[978], p[979], p[97], p[980], p[981], p[982], p[983], p[984], p[985],
p[986], p[987], p[988], p[989], p[98], p[990], p[991], p[992], p[993],
p[994], p[995], p[996], p[997], p[998], p[999], p[99], p[9] triggered
hat{ESS} warnings.
Small empirical effective sample sizes result in imprecise Markov chain
Monte Carlo estimators.
```

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

```
base_samples = util.filter_expectands(samples, ['alpha', 'beta'], True)
util.check_all_expectand_diagnostics(base_samples)
```

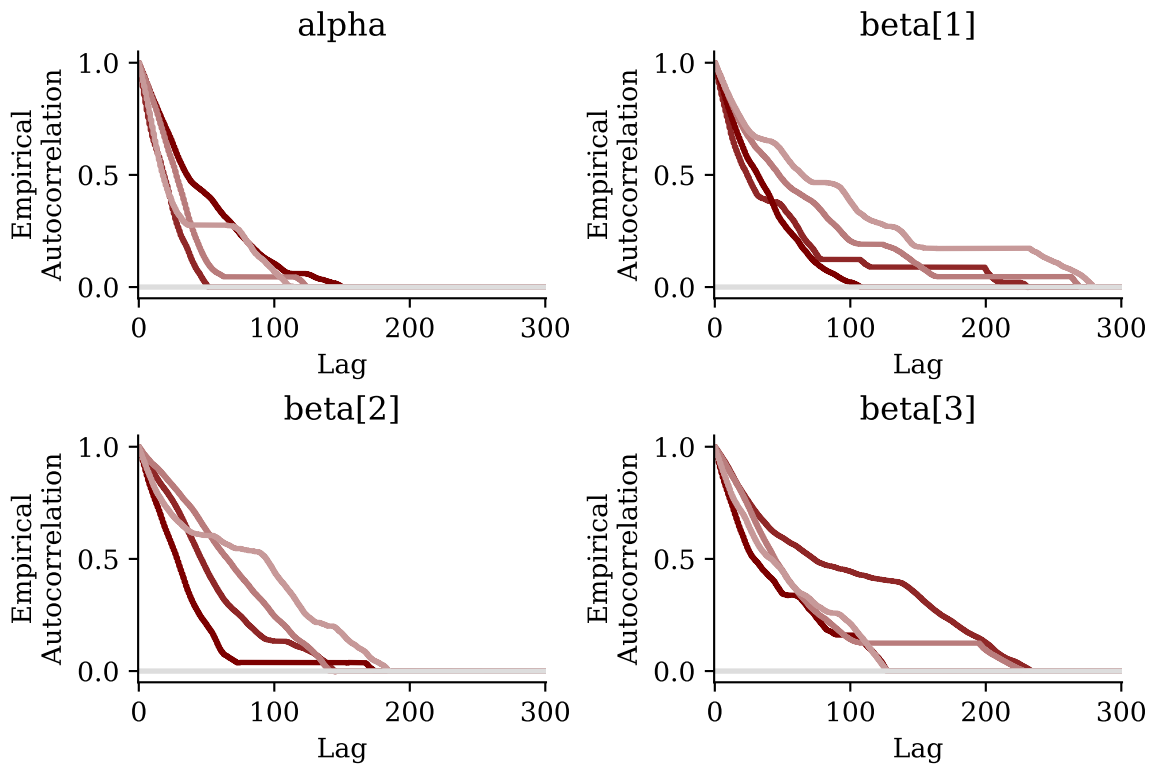
```
alpha:
  Split hat{R} (1.201) exceeds 1.1.
  Chain 1: hat{ESS} (25.7) is smaller than desired (100).
  Chain 2: hat{ESS} (10.8) is smaller than desired (100).
  Chain 3: hat{ESS} (16.5) is smaller than desired (100).
  Chain 4: hat{ESS} (14.7) is smaller than desired (100).

beta[1]:
  Split hat{R} (1.512) exceeds 1.1.
  Chain 1: hat{ESS} (10.9) is smaller than desired (100).
  Chain 2: hat{ESS} (14.1) is smaller than desired (100).
  Chain 3: hat{ESS} (7.7) is smaller than desired (100).
  Chain 4: hat{ESS} (5.5) is smaller than desired (100).

beta[2]:
  Split hat{R} (1.260) exceeds 1.1.
  Chain 1: hat{ESS} (9.5) is smaller than desired (100).
```

```
  Chain 2: hat{ESS} (15.1) is smaller than desired (100).
  Chain 3: hat{ESS} (7.6) is smaller than desired (100).
  Chain 4: hat{ESS} (6.4) is smaller than desired (100).

beta[3]:
  Split hat{R} (2.019) exceeds 1.1.
  Chain 1: hat{ESS} (5.3) is smaller than desired (100).
  Chain 2: hat{ESS} (11.4) is smaller than desired (100).
  Chain 3: hat{ESS} (8.0) is smaller than desired (100).
  Chain 4: hat{ESS} (9.9) is smaller than desired (100).


Split Rhat larger than 1.1 suggests that at least one of the Markov
chains has not reached an equilibrium.

Small empirical effective sample sizes result in imprecise Markov chain
Monte Carlo estimators.
```

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.

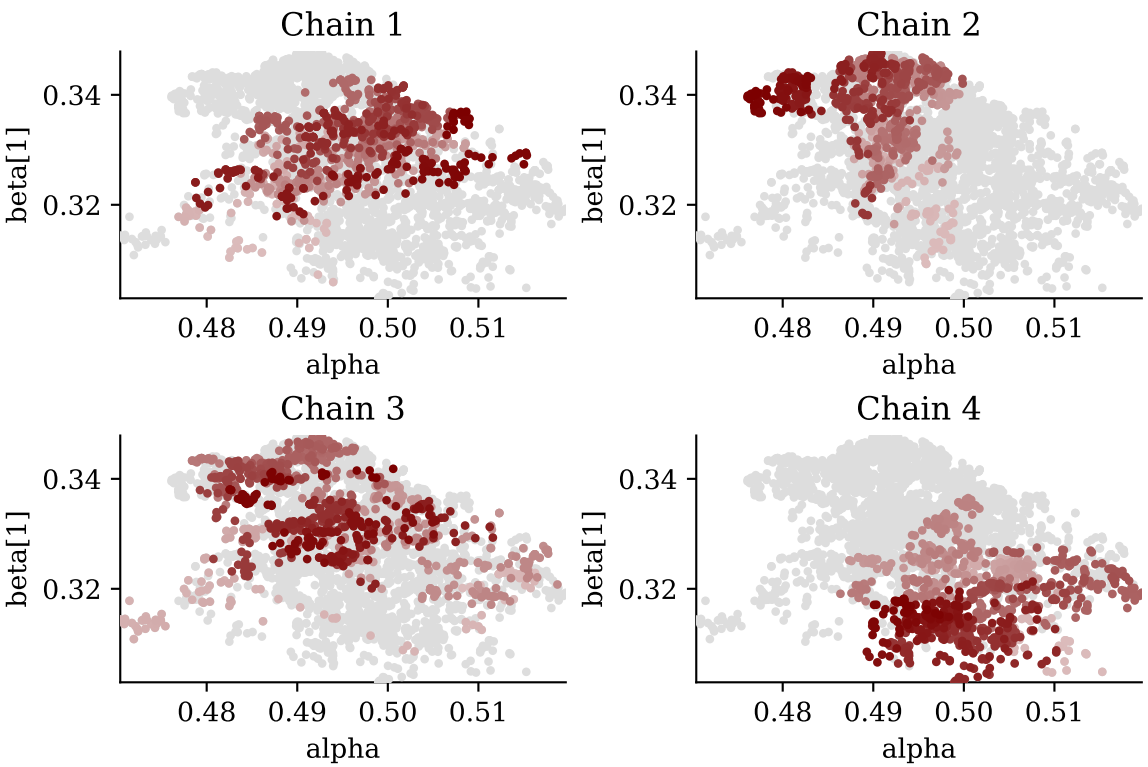
```
f, axarr = plot.subplots(2, 2, layout="constrained")

util.plot_empirical_correlogram(axarr[0, 0], samples['alpha'],
                                300, [-0.05, 1.05],  'alpha')
util.plot_empirical_correlogram(axarr[0, 1], samples['beta[1]'],
                                300, [-0.05, 1.05],  'beta[1]')
util.plot_empirical_correlogram(axarr[1, 0], samples['beta[2]'],
                                300, [-0.05, 1.05],  'beta[2]')
util.plot_empirical_correlogram(axarr[1, 1], samples['beta[3]'],
                                300, [-0.05, 1.05],  'beta[3]')

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.

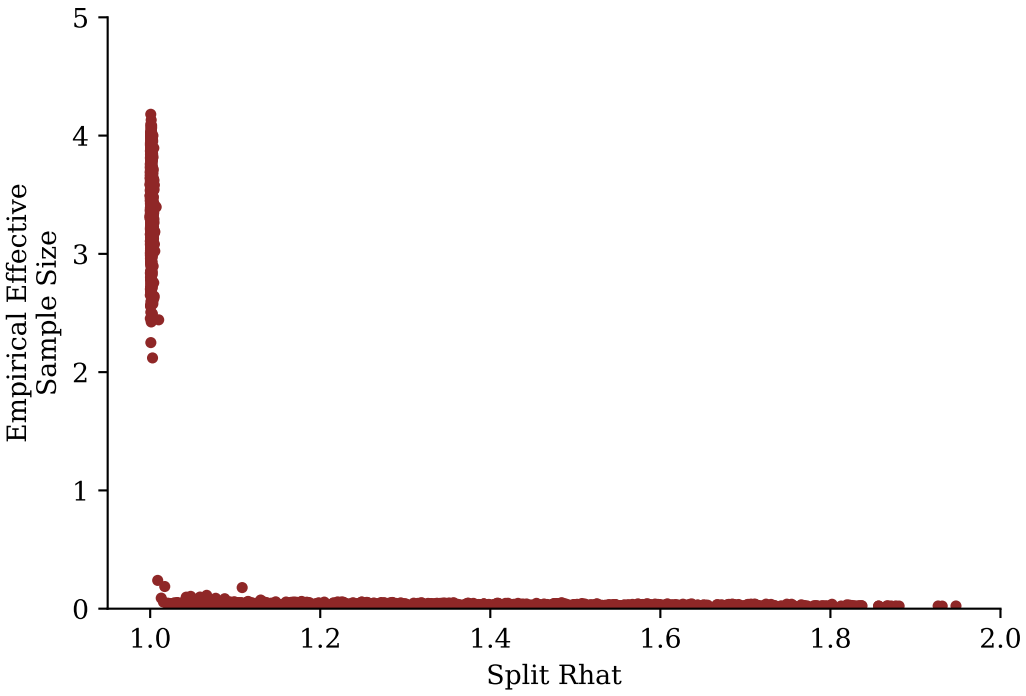
This is consistent with the constraint violations breaking the coherent, gradient-driven exploration of Hamiltonian Monte Carlo so that the Markov chains devolve into diffuse random walks. Indeed looking at the chain-separated pairs plots we see the spatial color continuity characteristic of a random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

```
util.plot_pairs_by_chain(samples['alpha'], 'alpha',
                         samples['beta[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 effective sample size. Specifically for each expectand we plot split $\hat{R}$ against we use the smallest empirical effective sample size of the four Markov chains.

```
rhats = util.compute_split_rhats(samples)
min_ess_hats = util.compute_min_ess_hats(samples)

plot.scatter(rhats, min_ess_hats, color=dark, s=10)
plot.gca().set_xlim([0.95, 2])
plot.gca().set_xlabel("Split Rhat")
plot.gca().set_ylim([0, 5])
plot.gca().set_ylabel("Empirical Effective\nSample Size")
plot.gca().spines["top"].set_visible(False)
plot.gca().spines["right"].set_visible(False)

plot.show()
```



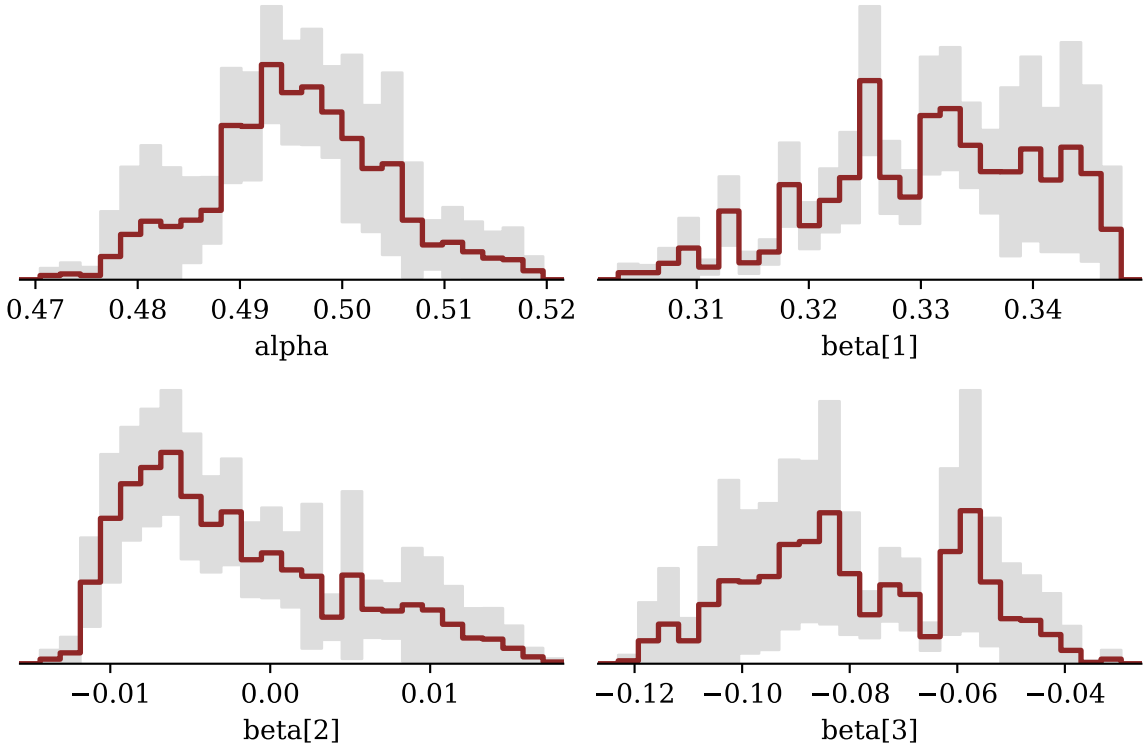
Every expectand with a large split $\hat{R}$s also exhibits a particularly small minimum empirical effective sample size, confirming that the latter are due to our Markov chains not containing enough information.

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 gray bands visualizing twice the standard error around the bin probability estimates in dark red.

```
f, axarr = plot.subplots(2, 2, layout="constrained")

util.plot_expectand_pushforward(axarr[0, 0], samples['alpha'],
                                25, display_name='alpha')
util.plot_expectand_pushforward(axarr[0, 1], samples['beta[1]'],
                                25, display_name='beta[1]')
util.plot_expectand_pushforward(axarr[1, 0], samples['beta[2]'],
                                25, display_name='beta[2]')
util.plot_expectand_pushforward(axarr[1, 1], samples['beta[3]'],
                                25, display_name='beta[3]')
```

```
plot.show()
```

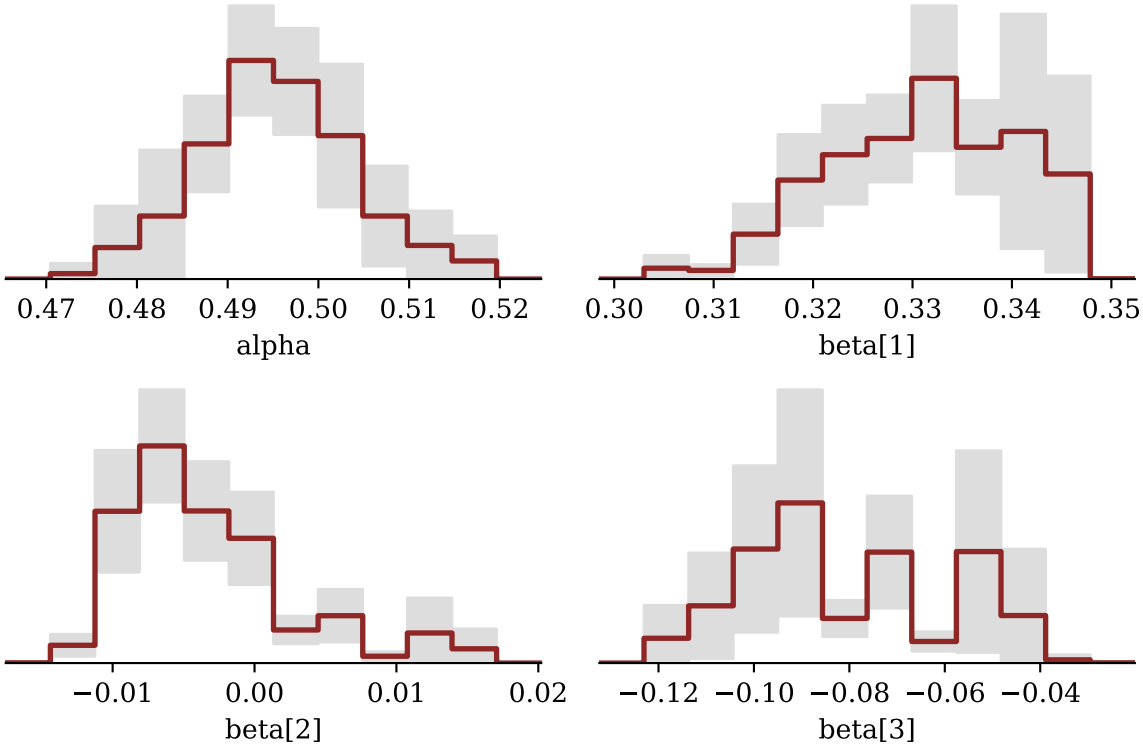


If we look at the central estimates alone we might convince ourselves of all kinds of interesting structure. For example potential multi-modality in `alpha` and `beta[2]` and platykurticity in `beta[1]` and `beta[3]`. These structures, however, are all within the scope of the relatively large standard 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, layout="constrained")

util.plot_expectand_pushforward(axarr[0, 0], samples['alpha'],
                                10, display_name='alpha')
util.plot_expectand_pushforward(axarr[0, 1], samples['beta[1]'],
                                10, display_name='beta[1]')
util.plot_expectand_pushforward(axarr[1, 0], samples['beta[2]'],
                                10, display_name='beta[2]')
util.plot_expectand_pushforward(axarr[1, 1], samples['beta[3]'],
```

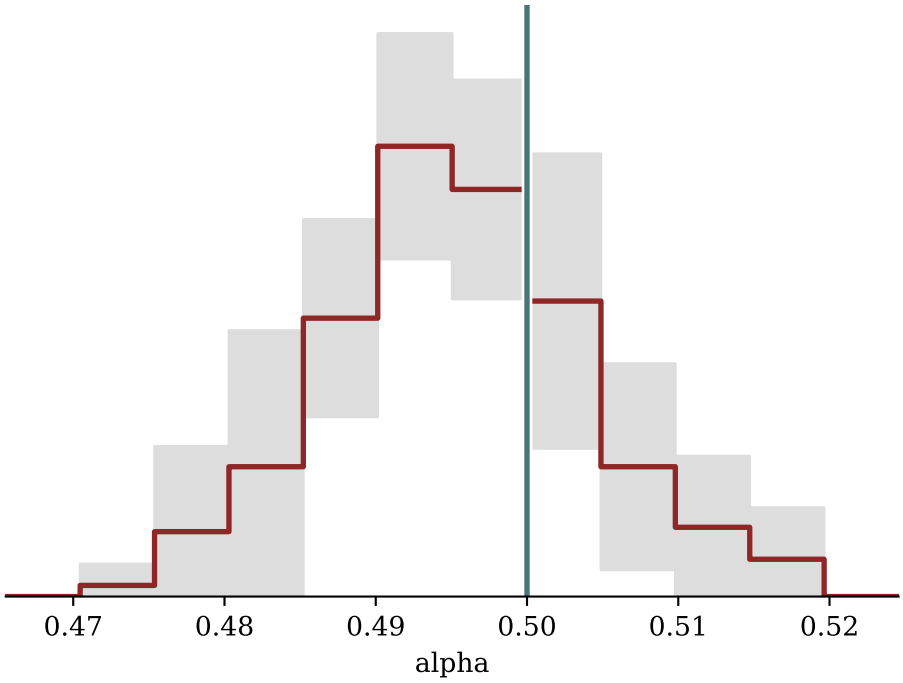
```
                                  10, display_name='beta[3]')

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.

The `plot_expectand_pushforward` can also overlay a baseline value for comparison, for example when comparing posterior inferences to the ground truth in simulation studies.

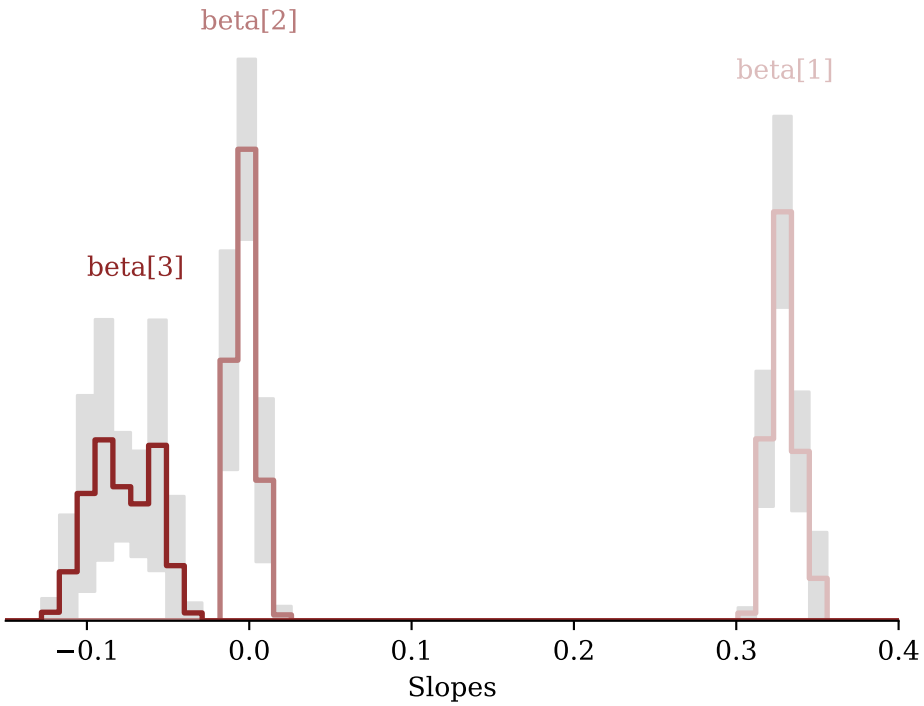
```
util.plot_expectand_pushforward(plot.gca(), samples['alpha'],
                                10, display_name="alpha",
                                baseline=0.5,
                                baseline_color=util.mid_teal)
plot.show()
```



Moreover the expectand pushforward histograms can be plotted on top of each other for a more direct comparison.

```
util.plot_expectand_pushforward(plot.gca(), samples['beta[1]'],
                                50, flim=[-0.15, 0.4],
                                ylim=[0, 60],
                                display_name="Slopes",
                                color=util.light)
plot.gca().text(0.3, 55, "beta[1]", color=util.light)

util.plot_expectand_pushforward(plot.gca(), samples['beta[2]'],
                                50, flim=[-0.15, 0.4],
                                color=util.mid,
                                border="#BBBBBB",
                                border_opacity=0.5,
                                add=True)
plot.gca().text(-0.03, 60, "beta[2]", color=util.mid)

util.plot_expectand_pushforward(plot.gca(), samples['beta[3]'],
                                50, flim=[-0.15, 0.4],
                                color=util.dark,
                                border="#BBBBBB",
```

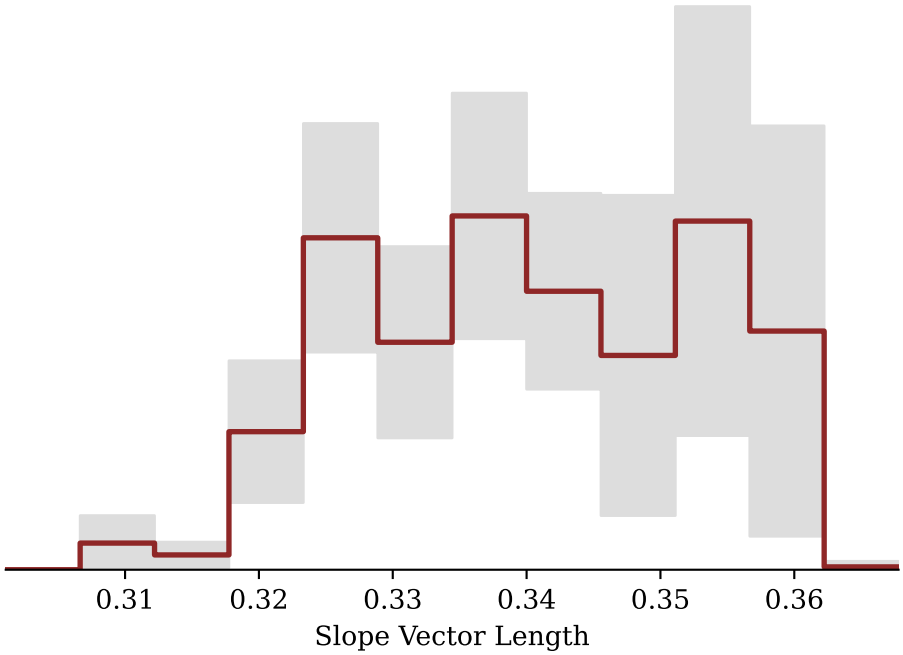
```
                                    border_opacity=0.5,
                                    add=True)
plot.gca().text(-0.1, 35, "beta[3]", color=util.dark)

plot.show()
```



Finally if we want to explore the pushforward posterior distribution of other expectands that have not already been evaluated in the `Stan` program then we need to evaluate them ourselves.

```
def euclidean_length(x):
  return math.sqrt(numpy.dot(x, x))

beta_names = util.name_array('beta', [data['M']])

pushforward_samples = \
  util.eval_expectand_pushforward(samples, euclidean_length,
                                  {'x': beta_names})

util.plot_expectand_pushforward(plot.gca(), pushforward_samples, 10,
```

```
                                    display_name="Slope Vector Length")
plot.show()
```



License

Original Computing Environment

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

```
Last updated: 2024-09-08T12:45:52.300758-04:00

Python implementation: CPython
Python version       : 3.9.6
IPython version      : 8.16.1

Compiler    : Clang 12.0.0 (clang-1200.0.32.29)
OS          : Darwin
Release     : 23.4.0
Machine     : x86_64
Processor   : i386
CPU cores   : 16
Architecture: 64bit
```

```
print(watermark(packages="matplotlib,numpy,stan,scipy"))
```

```
matplotlib: 3.8.0
numpy     : 1.26.1
stan      : 3.9.0
scipy     : 1.11.3
```

Stan
Program 1 `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
  array[N] real y;      // 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')');
  }
}
```

Stan
Program 2 `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

  array[N] int<lower=0, upper=1> y; // 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;
  array[N] int y_pred = bernoulli_rng(p);
}
```