Markov Chain Monte Carlo Visualization Functions

Michael Betancourt

2024-05-01

Table of contents

In this note I will review a suite of `python` functions that implement various visualizations of probabilistic behavior using the output of a Markov chain Monte Carlo algorithm.

Most of these visualizations utilize nested quantile intervals to visualize one-dimensional push-forward behavior as described in Chapter 7, Section 5 of my probability theory material. The individual quantiles are consistently estimated as the empirical average of the empirical quantiles derived from individual Markov chains. Because they do not communicate the quantile estimator errors these visualizations can be misleading if the Markov chains do not contain enough information.

1 Initial Setup

First and foremost we have to set up our local `python` environment.

```
import matplotlib
import matplotlib.pyplot as plot
plot.rcParams['figure.figsize'] = [5, 2.5]
plot.rcParams['figure.dpi'] = 100
plot.rcParams['font.family'] = "Serif"

import numpy
import scipy.stats as stats
import math

import json
```

This includes loading up `pystan`.

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

import stan
```

Finally we'll load my recommended Markov chain Monte Carlo analysis tools and the visualization functions themselves.

```
import mcmc_analysis_tools_pystan3 as util
```

```
import mcmc_visualization_tools as putil
```

2 One-Dimensional Baseline Function

Our first example is one-dimensional curve-fitting, i.e. regression, model with a linear baseline function,

$$p(y_n \mid x_n, \alpha, \beta, \sigma) = \text{normal}(y_n \mid \alpha + \beta\, x_n, \sigma).$$

2.1 Data Exploration

What is data if not an opportunity to explore?

```
with open("data/uni_data.json","r") as infile:
  data = json.load(infile)
```

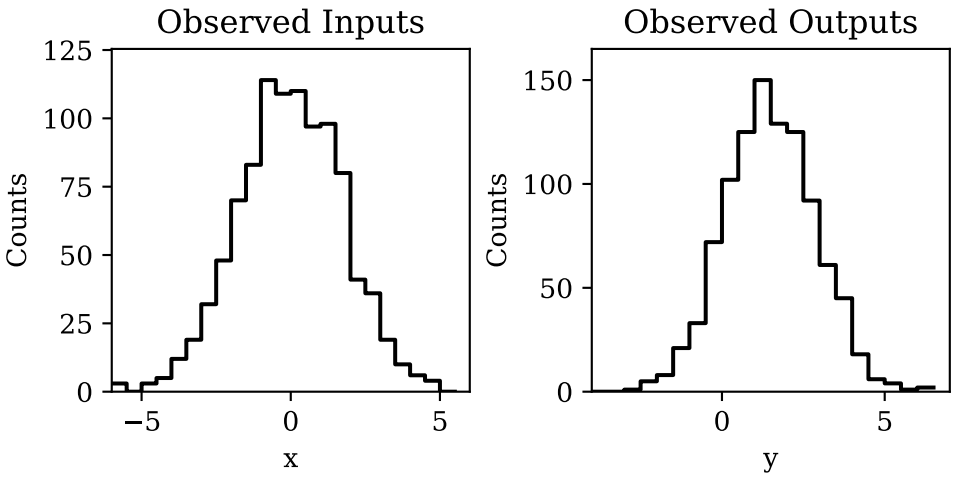
The `plot_line_hist` function constructs a histogram but then plots only its outline, without lines separating the interior histogram bins. Here we can plot histograms summarizing the observed inputs

$$\{\tilde{x}_1, \ldots, \tilde{x}_n, \ldots, \tilde{x}_N\}$$

and the observed outputs,

$$\{\tilde{y}_1, \ldots, \tilde{y}_n, \ldots, \tilde{y}_N\}.$$

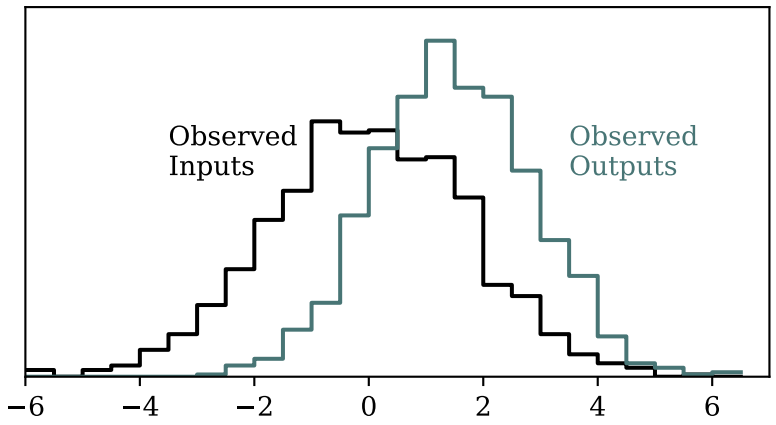
```
f, axarr = plot.subplots(1, 2, layout="constrained")
putil.plot_line_hist(axarr[0], data['x'], -6, 6, 0.5,
                     xlabel="x", title="Observed Inputs")
putil.plot_line_hist(axarr[1], data['y'], -4, 7, 0.5,
                     xlabel="y", title="Observed Outputs")
plot.show()
```



This presentation is a bit cleaner than conventional histogram plots, especially when there is no ambiguity about the binning. That said the implementation here is a bit rigid in that it only allows for uniform bin widths.

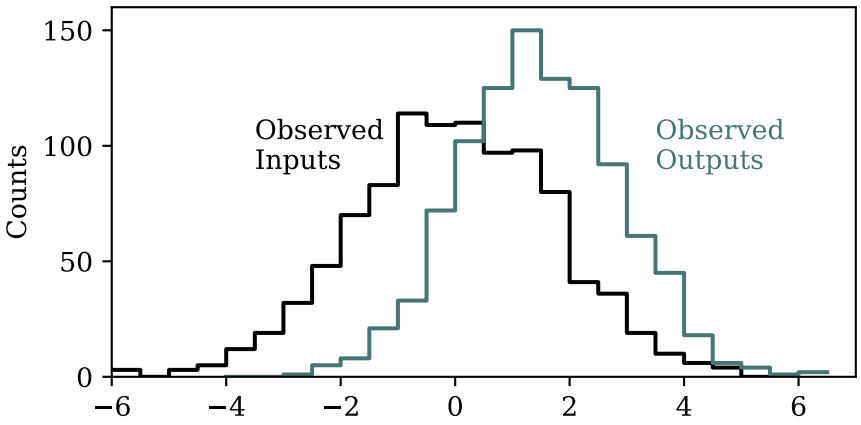
A key advantage of reducing a histogram to its outline is that it is much easier to overlay multiple histograms on top of each other without compromising legibility. The `plot_line_hists` function constructs and then overlays two histograms with the same binning.

```
putil.plot_line_hists(plot.gca(), data['x'], data['y'], -6, 7, 0.5)
plot.gca().text(-3.5, 90, "Observed\nInputs", color="black")
plot.gca().text(3.5, 90, "Observed\nOutputs", color=putil.mid_teal)
plot.show()
```



We can also use the `add` argument of `plot_line_hist` to overlay multiple histogram outlines onto an existing axis.

```
putil.plot_line_hist(plot.gca(), data['x'], -6, 6, 0.5,
                     col="black", add=True)
putil.plot_line_hist(plot.gca(), data['y'], -4, 7, 0.5,
                     col=putil.mid_teal, add=True)

plot.gca().text(-3.5, 90, "Observed\nInputs", color="black")
plot.gca().text(3.5, 90, "Observed\nOutputs", color=putil.mid_teal)

plot.gca().set_xlim([-6, 7])
plot.gca().set_xlabel("")
plot.gca().set_ylim([0, 160])
plot.gca().set_ylabel("Counts")

plot.show()
```



2.2 Prior Checks

Here we'll be exceptionally thorough and start with an investigation of the prior model and its consequences. In addition to the individual parameters we'll look at the prior behavior of baseline function and the prior predictive distribution along a grid of inputs defined by the `x_grid` array.

Note that I'm using a less-aggressive step size adaptation here because the half-normal prior model for σ results in an slightly awkward tail for the unconstrained $\log(\sigma)$ values that can be a bit difficult to navigate.

```
data['N_grid'] = 1000
data['x_grid'] = numpy.arange(-6, 6, 12 / data['N_grid'])

with open('stan_programs/uni_prior_model.stan', 'r') as file:
  stan_program = file.read()
model = stan.build(stan_program, random_seed=5838299, data=data)
fit = model.sample(num_samples=1024, refresh=0, delta=0.9)
```

```
Building...
```

Of course we always consult our diagnostics first to make sure that our Markov chains, and hence any visualization we derive from them, accurately characterize the exact target distribution, in this case the prior distribution of our model.

Stan
Program 1 uni_prior_model.stan

```
data {
  int<lower=1> N;
  vector[N] x; // Observed inputs
  vector[N] y; // Observed outputs

  int<lower=1> N_grid; // Number of grid points for quantifying functional behavior
  vector[N_grid] x_grid; // Grid points for quantifying functional behavior
}

parameters {
  real alpha;
  real beta;
  real<lower=0> sigma;
}

model {
  alpha ~ normal(0, 3 / 2.32);
  beta ~ normal(0, 3 / 2.32);
  sigma ~ normal(0, 1 / 2.57);
}

generated quantities {
  vector[N_grid] f_grid = alpha + beta * x_grid;
  array[N_grid] real y_pred_grid = normal_rng(f_grid, sigma);
}
```

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

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

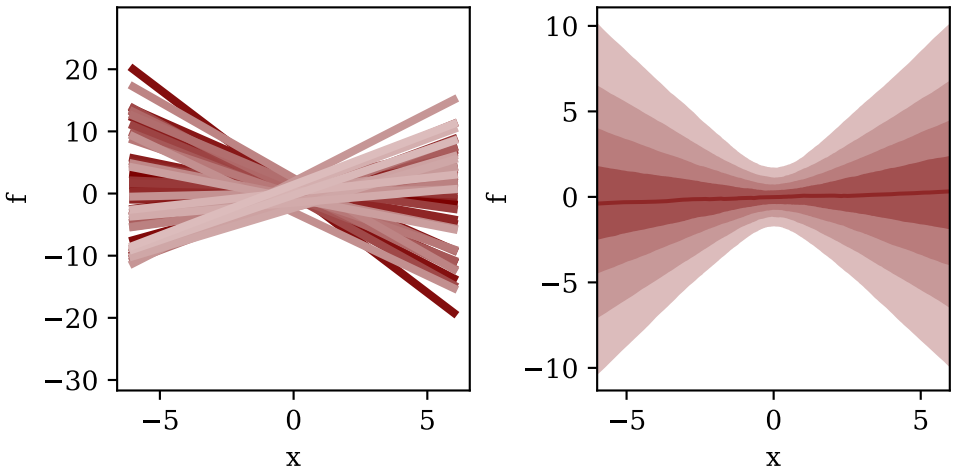
```
All Hamiltonian Monte Carlo diagnostics are consistent with accurate
Markov chain Monte Carlo.
```

```
All expectands checked appear to be behaving well enough for reliable
Markov chain Monte Carlo estimation.
```

We can visualize the probability distribution of baseline functions in two ways. Firstly we can plot a subset of baseline function configurations. Secondly we can plot nested quantile intervals that quantify the marginal behavior of the function output at each input. Neither of these visualizations fully characterize the probabilistic behavior but together they capture the most important features.

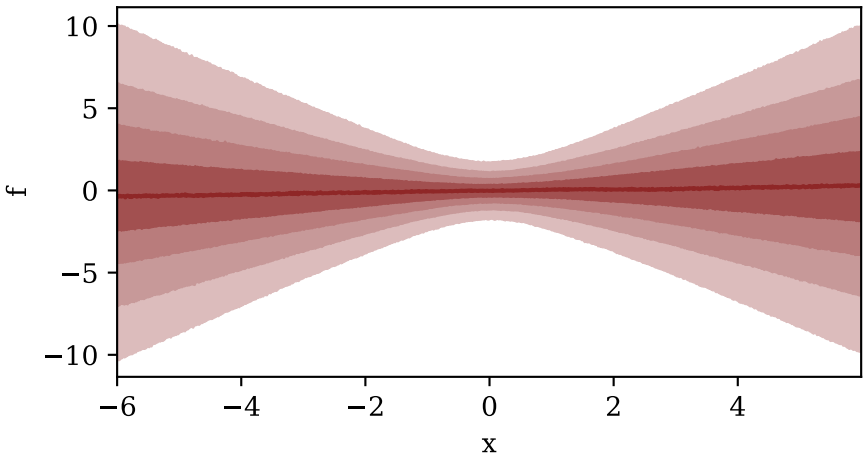
The `plot_realizations` function plots a selection of values corresponding to the `f_names` array against `data$x_grid` while the `plot_conn_pushforward_quantiles` function plots nested quantile intervals of those values for each element of `data['x_grid']`. Here "conn" refers to "connected" as the individual marginal quantiles are connected into continuous polygons.

```
f, axarr = plot.subplots(1, 2, layout="constrained")

f_names = [ f'f_grid[{n + 1}]' for n in range(data['N_grid']) ]
putil.plot_realizations(axarr[0], samples, f_names, data['x_grid'],
                        xlabel="x", ylabel="f")
putil.plot_conn_pushforward_quantiles(axarr[1], samples,
                                      f_names, data['x_grid'],
                                      xlabel="x", ylabel="f")

plot.show()
```



Finally let's use the `plot_conn_pushforward_quantiles` function to plot nested quantile intervals of the conditional prior predictive behavior at each element of `data$x_grid`.

```
pred_names = [ f'y_pred_grid[{n + 1}]' for n in range(data['N_grid']) ]
putil.plot_conn_pushforward_quantiles(plot.gca(), samples,
                                      pred_names, data['x_grid'],
                                      xlabel="x", ylabel="f")
plot.show()
```



2.3 Posterior Inference

Having thoroughly investigated our prior model and its consequences and not found any undesired behavior we can move on to constructing posterior inferences.

```
with open('stan_programs/uni_full_model.stan', 'r') as file:
  stan_program = file.read()
model = stan.build(stan_program, random_seed=5838299, data=data)
fit = model.sample(num_samples=1024, refresh=0)
```

```
Building...
```

There are no signs of trouble from the computational diagnostics.

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

samples = util.extract_expectands(fit)
base_samples = util.filter_expectands(samples,
```

```
                                        ['alpha', 'beta', 'sigma'])
util.check_all_expectand_diagnostics(base_samples)
```

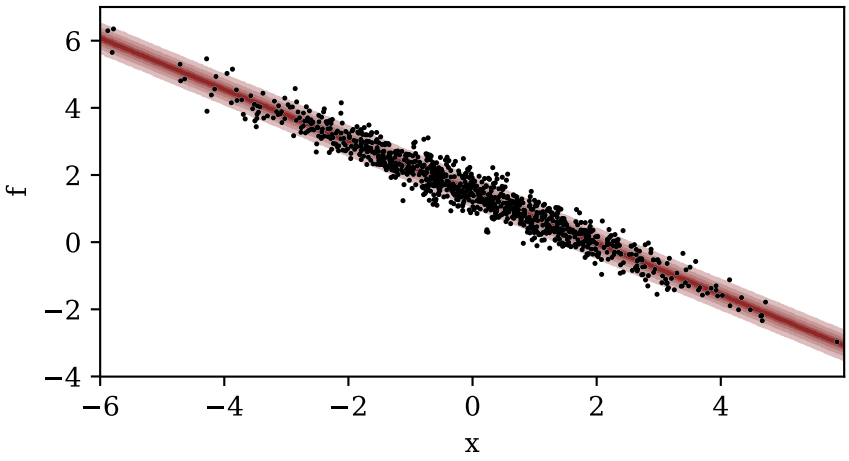
```
All Hamiltonian Monte Carlo diagnostics are consistent with accurate
Markov chain Monte Carlo.

All expectands checked appear to be behaving well enough for reliable
Markov chain Monte Carlo estimation.
```

Before examining any posterior inferences, however, we need to validate that our model is adequately capturing the relevant features of the observed data. For this one-dimensional baseline function model we can implement an informative retrodictive check by comparing the conditional posterior predictive distributions at each input,

$$p(y \mid x, \tilde{x}_1, \tilde{y}_1, \ldots, \tilde{x}_N, \tilde{y}_N),$$

to the observed input-output pairs, (x_n, y_n).

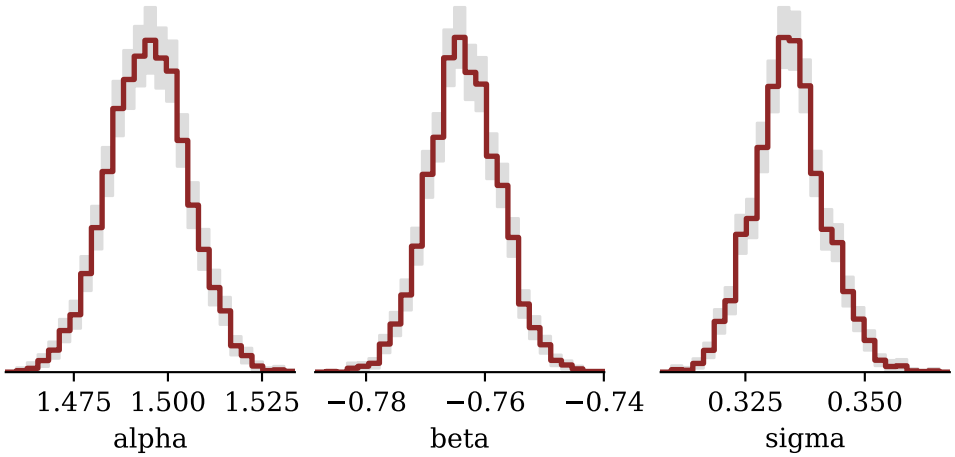
```
pred_names = [ f'y_pred_grid[{n + 1}]' for n in range(data['N_grid']) ]
putil.plot_conn_pushforward_quantiles(plot.gca(), samples,
                                      pred_names, data['x_grid'],
                                      xlabel="x", ylabel="f")
plot.scatter(data['x'], data['y'], s=1, color="white", zorder=4)
plot.scatter(data['x'], data['y'], s=0.8, color="black", zorder=4)

plot.show()
```



Fortunately there are no signs of tension between the posterior predictive distributional behaviors and the observed behaviors. Confident in the adequacy of our model we can move onto visualizing posterior inferences.

For example we can visualize the pushforward, or marginal, probability distributions for each parameter. Note that the `plot_expectand_pushforward` function is already part of my Markov chain Monte Carlo analysis tools and not one of the visualization functions being introduced here.

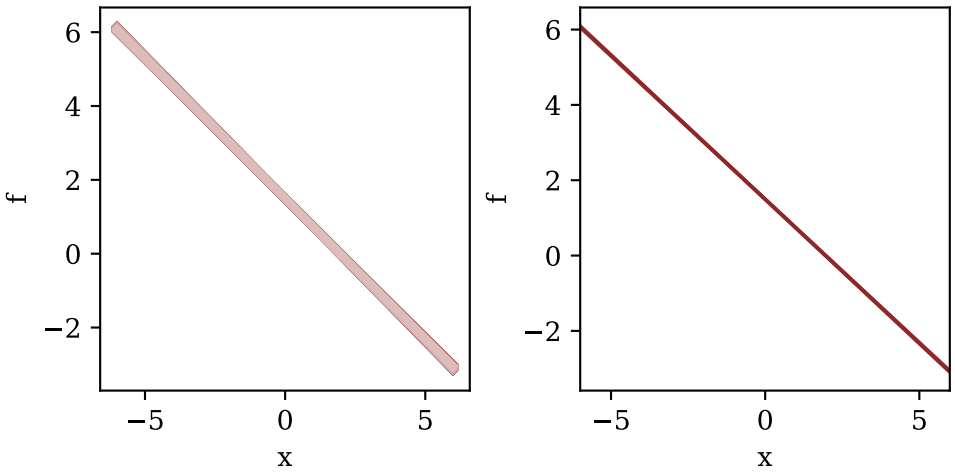
```
f, axarr = plot.subplots(1, 3, layout="constrained")

util.plot_expectand_pushforward(axarr[0], samples['alpha'],
                                25, display_name="alpha")

util.plot_expectand_pushforward(axarr[1], samples['beta'],
                                25, display_name="beta")

util.plot_expectand_pushforward(axarr[2], samples['sigma'],
                                25, display_name="sigma")

plot.show()
```



Communicating the posterior behavior of the baseline function, however, is facilitated with the new visualization functions.

```
f, axarr = plot.subplots(1, 2, layout="constrained")

f_names = [ f'f_grid[{n + 1}]' for n in range(data['N_grid']) ]
```

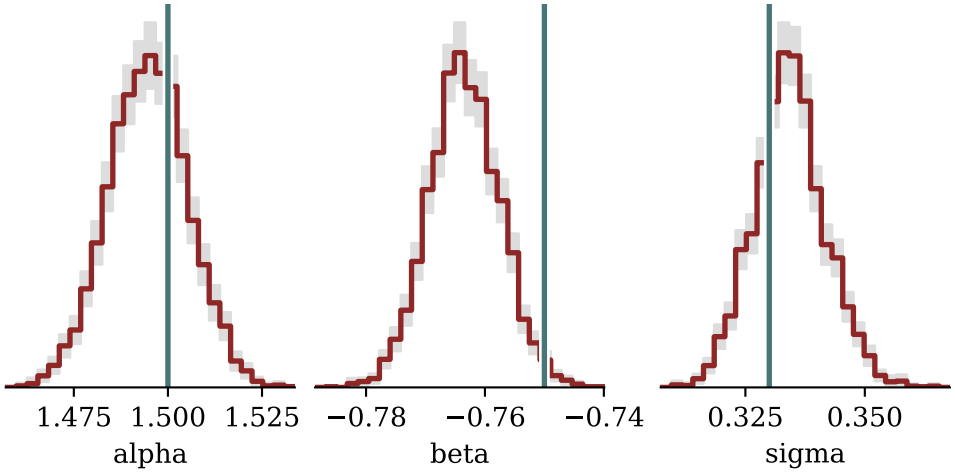
```
putil.plot_realizations(axarr[0], samples, f_names, data['x_grid'],
                        xlabel="x", ylabel="f")
putil.plot_conn_pushforward_quantiles(axarr[1], samples,
                                      f_names, data['x_grid'],
                                      xlabel="x", ylabel="f")

plot.show()
```



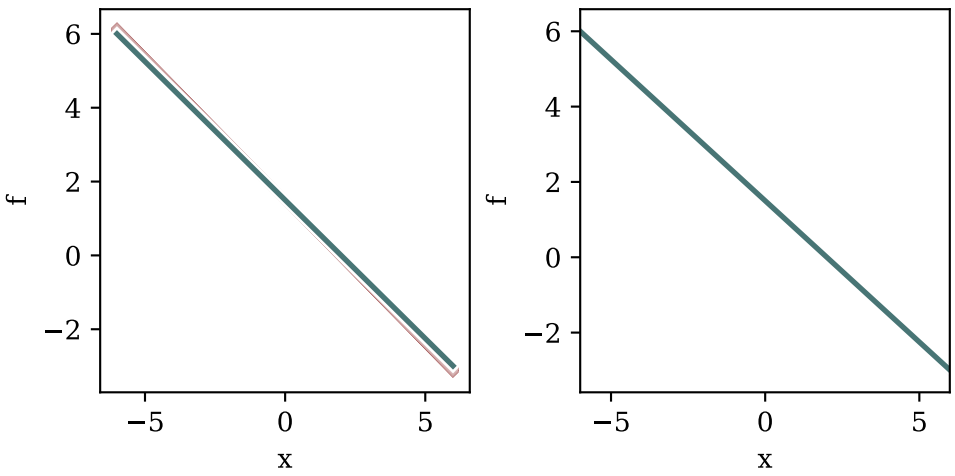
Conveniently all of these visualization functions feature optional arguments for baseline behavior which allows us to compare our posterior inferences to the true behavior when it is known, for example in simulation studies.

```
true_alpha = 1.5
true_beta = -0.75
true_sigma = 0.33

f, axarr = plot.subplots(1, 3, layout="constrained")

util.plot_expectand_pushforward(axarr[0], samples['alpha'],
                                25, display_name="alpha",
                                baseline=true_alpha,
                                baseline_color=putil.mid_teal)

util.plot_expectand_pushforward(axarr[1], samples['beta'],
                                25, display_name="beta",
                                baseline=true_beta,
                                baseline_color=putil.mid_teal)
```

```
util.plot_expectand_pushforward(axarr[2], samples['sigma'],
                                25, display_name="sigma",
                                baseline=true_sigma,
                                baseline_color=putil.mid_teal)

plot.show()
```

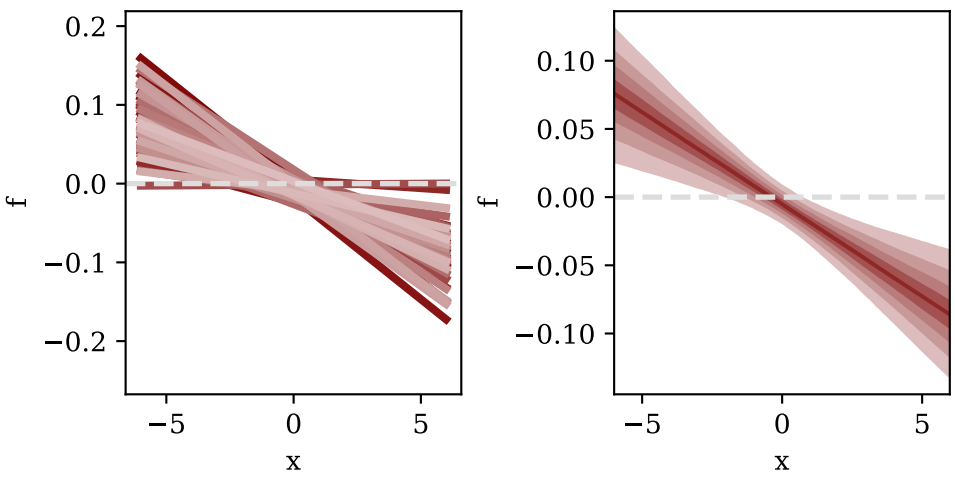



```
true_fs = [ true_alpha + true_beta * x for x in data['x_grid'] ]

f, axarr = plot.subplots(1, 2, layout="constrained")
putil.plot_realizations(axarr[0], samples,
                        f_names, data['x_grid'],
                        baseline_values=true_fs,
                        baseline_color=putil.mid_teal,
                        xlabel="x", ylabel="f")
putil.plot_conn_pushforward_quantiles(axarr[1], samples,
                                      f_names, data['x_grid'],
                                      baseline_values=true_fs,
                                      baseline_color=putil.mid_teal,
                                      xlabel="x", ylabel="f")

plot.show()
```



The `plot_realizations` and `plot_conn_pushforward_quantiles` functions also include `residual` arguments that allow us to directly visualize how the probabilistic behavior varies around the baseline values.

```
f, axarr = plot.subplots(1, 2, layout="constrained")
putil.plot_realizations(axarr[0], samples,
                        f_names, data['x_grid'],
                        baseline_values=true_fs,
                        residual=True,
                        xlabel="x", ylabel="f")
putil.plot_conn_pushforward_quantiles(axarr[1], samples,
                                      f_names, data['x_grid'],
                                      residual=True,
                                      baseline_values=true_fs,
                                      xlabel="x", ylabel="f")

plot.show()
```



3 Multi-Dimensional Baseline Function

Now that we're warmed up let's consider a three-dimensional curve-fitting model with a quadratic baseline function,

$$p(y_n \mid \mathbf{x}_n, \alpha, \beta, \sigma) = \text{normal}(y_n \mid \beta_0 + \beta^T \cdot \mathbf{x} + \mathbf{x}^T \cdot \mathbf{B} \cdot \mathbf{x}, \sigma),$$

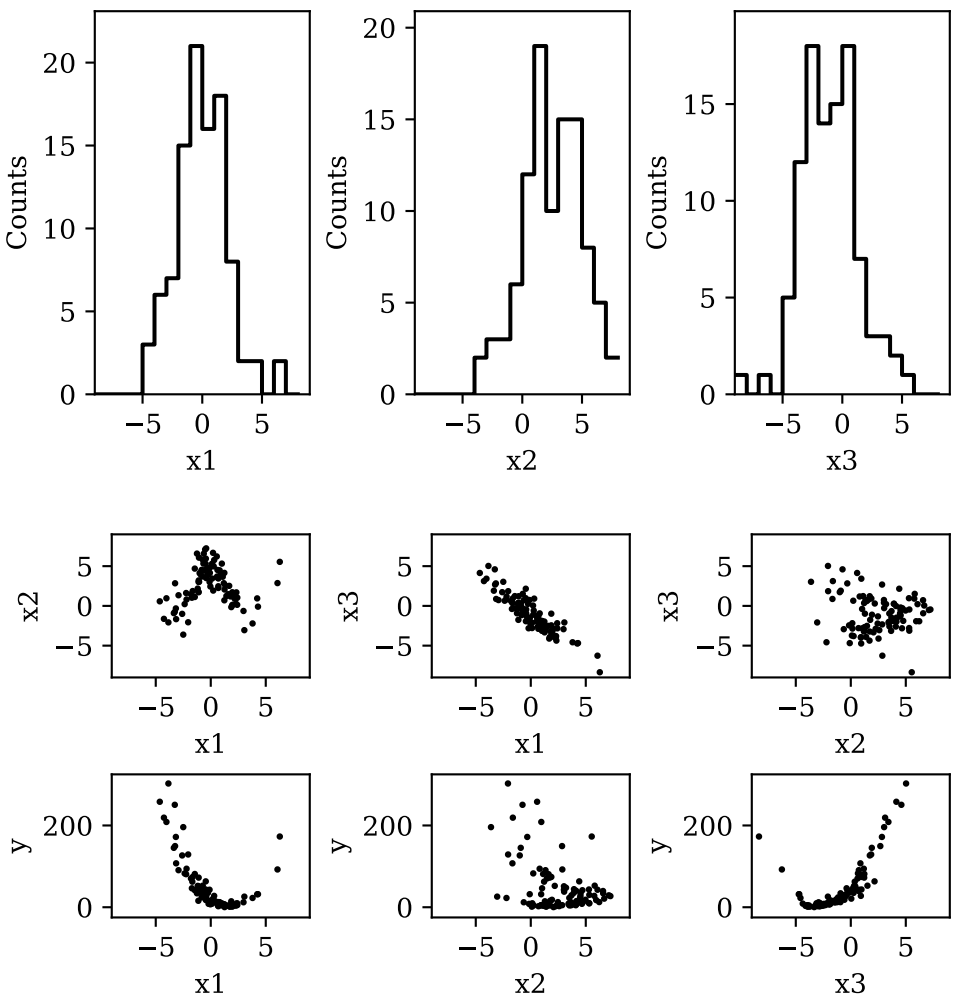
where $\mathbf{B}$ is a positive-definite matrix whose three diagonal elements are organized into the vector β_d and three off-diagonal elements are organized into the vector β_o.

3.1 Plot Data

The `plot_line_hist` allows us to cleanly visualize each component of the observed inputs.

```
with open("data/multi_data.json","r") as infile:
  data = json.load(infile)
data['X'] = numpy.asarray(data['X'])


f, axarr = plot.subplots(1, 3, layout="constrained")

putil.plot_line_hist(axarr[0], data['X'][:,0], -9, 9, 1, xlabel="x1")
putil.plot_line_hist(axarr[1], data['X'][:,1], -9, 9, 1, xlabel="x2")
putil.plot_line_hist(axarr[2], data['X'][:,2], -9, 9, 1, xlabel="x3")

plot.show()
```

```
f, axarr = plot.subplots(2, 3, layout="constrained")

axarr[0, 0].scatter(data['X'][:,0], data['X'][:,1], color="black", s=2)
axarr[0, 0].set_xlim([-9, 9])
axarr[0, 0].set_xlabel("x1")
axarr[0, 0].set_ylim([-9, 9])
axarr[0, 0].set_ylabel("x2")

axarr[0, 1].scatter(data['X'][:,0], data['X'][:,2], color="black", s=2)
axarr[0, 1].set_xlim([-9, 9])
axarr[0, 1].set_xlabel("x1")
axarr[0, 1].set_ylim([-9, 9])
axarr[0, 1].set_ylabel("x3")

axarr[0, 2].scatter(data['X'][:,1], data['X'][:,2], color="black", s=2)
axarr[0, 2].set_xlim([-9, 9])
axarr[0, 2].set_xlabel("x2")
axarr[0, 2].set_ylim([-9, 9])
axarr[0, 2].set_ylabel("x3")

axarr[1, 0].scatter(data['X'][:,0], data['y'], color="black", s=2)
axarr[1, 0].set_xlim([-9, 9])
axarr[1, 0].set_xlabel("x1")
axarr[1, 0].set_ylim([-25, 325])
axarr[1, 0].set_ylabel("y")

axarr[1, 1].scatter(data['X'][:,1], data['y'], color="black", s=2)
axarr[1, 1].set_xlim([-9, 9])
axarr[1, 1].set_xlabel("x2")
axarr[1, 1].set_ylim([-25, 325])
axarr[1, 1].set_ylabel("y")

axarr[1, 2].scatter(data['X'][:,2], data['y'], color="black", s=2)
axarr[1, 2].set_xlim([-9, 9])
axarr[1, 2].set_xlabel("x3")
axarr[1, 2].set_ylim([-25, 325])
axarr[1, 2].set_ylabel("y")

plot.show()
```



3.2 Prior Checks

As before we'll first investigate the consequences of our prior model.

For a discussion of why the quadratic baseline model is implemented in this way see Section 2.3.2 of my Taylor regression modeling chapter.

```
with open('stan_programs/multi_prior_model.stan', 'r') as file:
  stan_program = file.read()
model = stan.build(stan_program, random_seed=5838299, data=data)
fit = model.sample(num_samples=1024, refresh=0)
```

```
Building...
```

Higher-dimensional probability distributions are no trouble for Hamiltonian Monte Carlo.

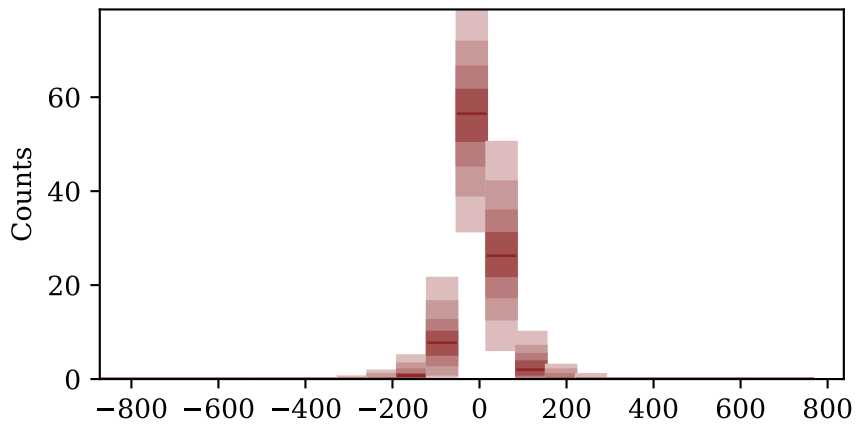
```
diagnostics = util.extract_hmc_diagnostics(fit)
util.check_all_hmc_diagnostics(diagnostics)

samples = util.extract_expectands(fit)
base_samples = util.filter_expectands(samples,
                                      ['beta0', 'beta1',
                                       'beta2_d', 'beta2_o',
                                       'sigma'],
                                      True)
util.check_all_expectand_diagnostics(base_samples)
```

```
All Hamiltonian Monte Carlo diagnostics are consistent with accurate
Markov chain Monte Carlo.

All expectands checked appear to be behaving well enough for reliable
Markov chain Monte Carlo estimation.
```

With a multi-dimensional input space we can no longer visualize the baseline functional behavior nor the conditional prior predictive behavior directly. We can, however, visualize many of its features.

For example we might consider the marginal behavior of the predicted outputs, regardless of the corresponding observed inputs. Here we'll summarize this marginal behavior with a histogram, and use the `plot_hist_quantiles` function to visualize the prior predictive distribution of the histogram counts.

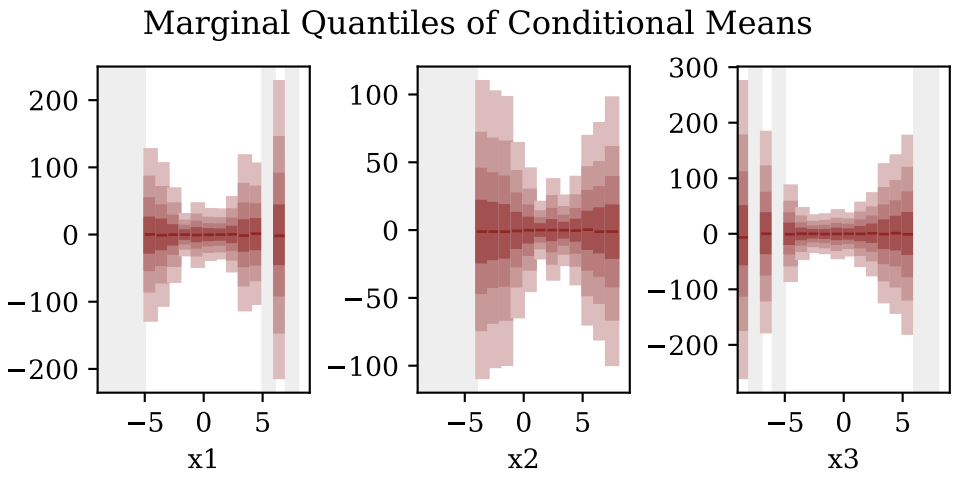
```
putil.plot_hist_quantiles(plot.gca(), samples, 'y_pred')
```



To capture the interactions between the predictive outputs and the observed input components we'll need a more sophisticated summary statistic. Here we'll use the empirical mean and medians of the predictive outputs within bins of each input component. For a detailed discussion of how this summary statistic is constructed see Section 2.5 of my Taylor regression modeling chapter.

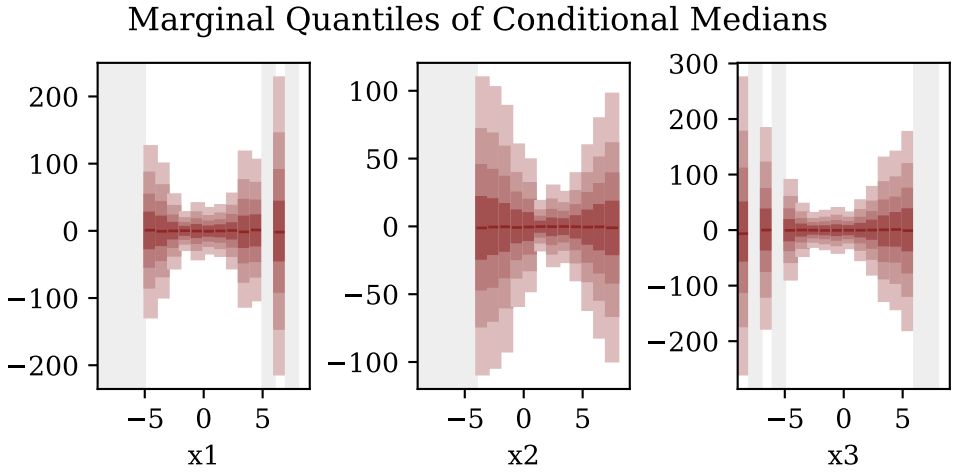
Conveniently the `plot_conditional_mean_quantiles` and `plot_conditional_median_quantiles` functions visualize the prior predictive behavior of these summary statistics.

```
f, axarr = plot.subplots(1, 3, layout="constrained")
f.suptitle("Marginal Quantiles of Conditional Means")

pred_names = [ f'y_pred[{n + 1}]' for n in range(data['N']) ]
putil.plot_conditional_mean_quantiles(axarr[0], samples, pred_names,
                                      data['X'][:,0], -9, 9, 1,
                                      xlabel="x1", ylabel="")
putil.plot_conditional_mean_quantiles(axarr[1], samples, pred_names,
                                      data['X'][:,1], -9, 9, 1,
                                      xlabel="x2", ylabel="")
putil.plot_conditional_mean_quantiles(axarr[2], samples, pred_names,
                                      data['X'][:,2], -9, 9, 1,
                                      xlabel="x3", ylabel="")
plot.show()
```




```
f, axarr = plot.subplots(1, 3, layout="constrained")
f.suptitle("Marginal Quantiles of Conditional Medians")

pred_names = [ f'y_pred[{n + 1}]' for n in range(data['N']) ]
```

```
putil.plot_conditional_median_quantiles(axarr[0], samples, pred_names,
                                        data['X'][:,0], -9, 9, 1,
                                        xlabel="x1", ylabel="")
putil.plot_conditional_median_quantiles(axarr[1], samples, pred_names,
                                        data['X'][:,1], -9, 9, 1,
                                        xlabel="x2", ylabel="")
putil.plot_conditional_median_quantiles(axarr[2], samples, pred_names,
                                        data['X'][:,2], -9, 9, 1,
                                        xlabel="x3", ylabel="")
plot.show()
```



Now we're ready to incorporate the observed data.

```
with open('stan_programs/multi_full_model.stan', 'r') as file:
  stan_program = file.read()
model = stan.build(stan_program, random_seed=5838299, data=data)
fit = model.sample(num_samples=1024, refresh=0)
```

```
Building...
```

Fortunately our computational fortune has persisted.

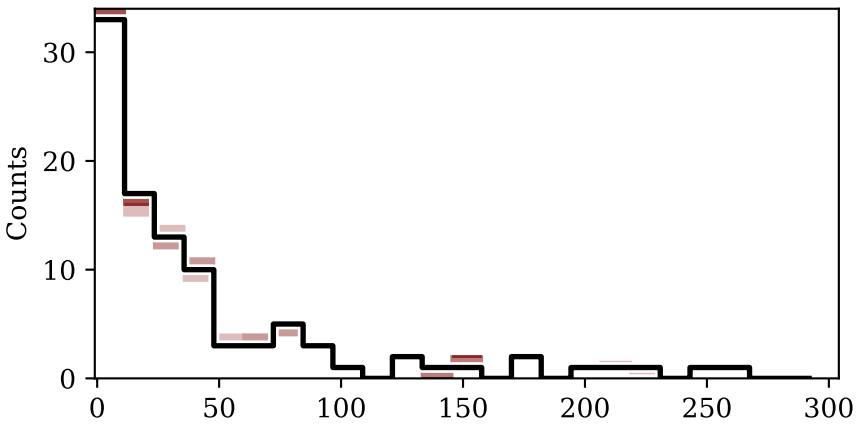
```
diagnostics = util.extract_hmc_diagnostics(fit)
util.check_all_hmc_diagnostics(diagnostics)

samples = util.extract_expectands(fit)
```

```
base_samples = util.filter_expectands(samples,
                                      ['beta0', 'beta1',
                                       'beta2_d', 'beta2_o',
                                       'sigma'],
                                      True)
util.check_all_expectand_diagnostics(base_samples)
```

```
All Hamiltonian Monte Carlo diagnostics are consistent with accurate
Markov chain Monte Carlo.

All expectands checked appear to be behaving well enough for reliable
Markov chain Monte Carlo estimation.
```

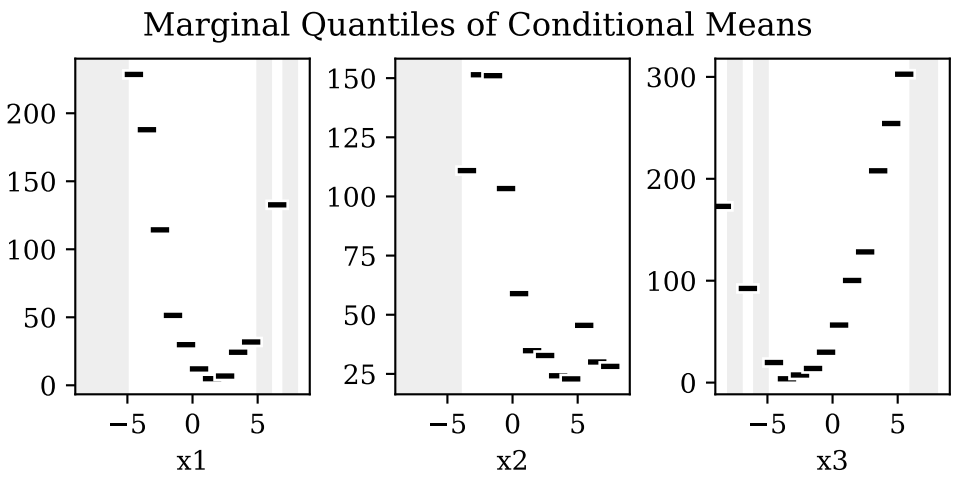
The summary statistics that we used above to implement our prior predictive checks are equally useful for implementing informative posterior retrodictive checks. Conveniently the visualization functions all feature `baseline_values` functions that we can use to visualize the observed behavior along with the posterior predictive behavior.

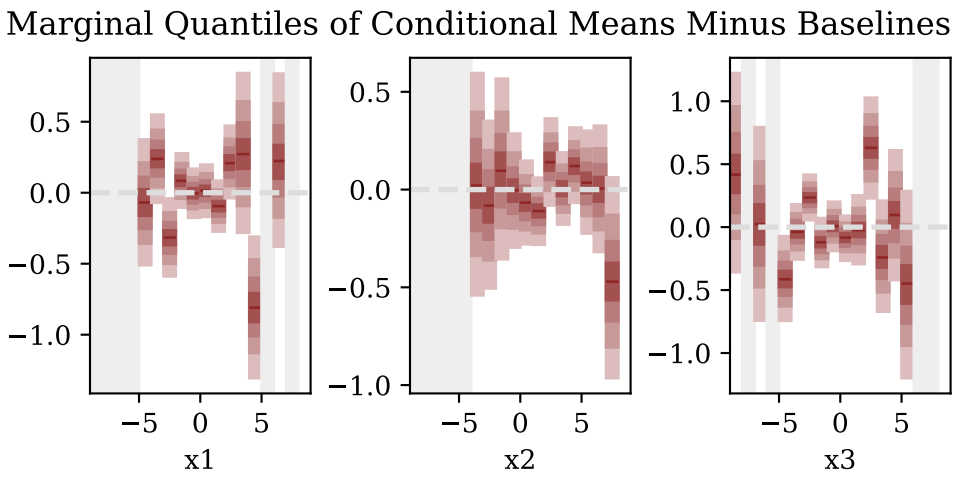
```
putil.plot_hist_quantiles(plot.gca(), samples, 'y_pred',
                          baseline_values=data['y'])
```



Additionally the `plot_conditional_mean_quantiles` and `plot_conditional_median_quantiles` functions feature a `residual` option that plots the posterior predictive behaviors relative to the baseline values. Any deviations from zero in these plots suggests retrodictive tension; here, however, there don't seem to be any problems.

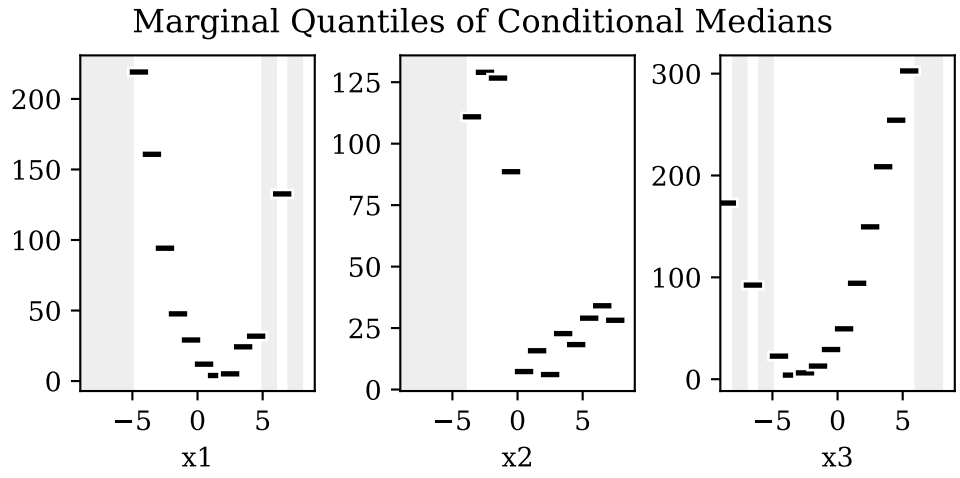
```
f, axarr = plot.subplots(1, 3, layout="constrained")
f.suptitle("Marginal Quantiles of Conditional Means")

pred_names = [ f'y_pred[{n + 1}]' for n in range(data['N']) ]
putil.plot_conditional_mean_quantiles(axarr[0], samples, pred_names,
                                      data['X'][:,0], -9, 9, 1,
                                      data['y'], xlabel="x1", ylabel="")
putil.plot_conditional_mean_quantiles(axarr[1], samples, pred_names,
                                      data['X'][:,1], -9, 9, 1,
                                      data['y'], xlabel="x2", ylabel="")
putil.plot_conditional_mean_quantiles(axarr[2], samples, pred_names,
                                      data['X'][:,2], -9, 9, 1,
                                      data['y'], xlabel="x3", ylabel="")
plot.show()

f, axarr = plot.subplots(1, 3, layout="constrained")
f.suptitle("Marginal Quantiles of Conditional Means Minus Baselines")

putil.plot_conditional_mean_quantiles(axarr[0], samples, pred_names,
                                      data['X'][:,0], -9, 9, 1,
                                      data['y'], residual=True,
                                      xlabel="x1", ylabel="")
putil.plot_conditional_mean_quantiles(axarr[1], samples, pred_names,
                                      data['X'][:,1], -9, 9, 1,
                                      data['y'], residual=True,
                                      xlabel="x2", ylabel="")
putil.plot_conditional_mean_quantiles(axarr[2], samples, pred_names,
                                      data['X'][:,2], -9, 9, 1,
                                      data['y'], residual=True,
                                      xlabel="x3", ylabel="")
plot.show()
```

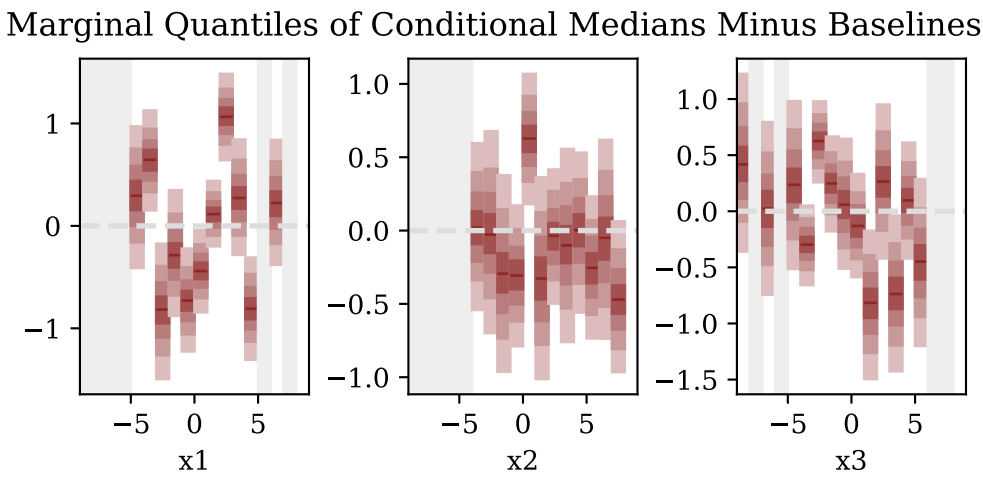





```
f, axarr = plot.subplots(1, 3, layout="constrained")
f.suptitle("Marginal Quantiles of Conditional Medians")

pred_names = [ f'y_pred[{n + 1}]' for n in range(data['N']) ]
putil.plot_conditional_median_quantiles(axarr[0], samples, pred_names,
                                        data['X'][:,0], -9, 9, 1,
                                        data['y'], xlabel="x1", ylabel="")
putil.plot_conditional_median_quantiles(axarr[1], samples, pred_names,
                                        data['X'][:,1], -9, 9, 1,
                                        data['y'], xlabel="x2", ylabel="")
putil.plot_conditional_median_quantiles(axarr[2], samples, pred_names,
                                        data['X'][:,2], -9, 9, 1,
                                        data['y'], xlabel="x3", ylabel="")
```

```
plot.show()

f, axarr = plot.subplots(1, 3, layout="constrained")
f.suptitle("Marginal Quantiles of Conditional Medians Minus Baselines")

putil.plot_conditional_median_quantiles(axarr[0], samples, pred_names,
                                        data['X'][:,0], -9, 9, 1,
                                        data['y'], residual=True,
                                        xlabel="x1", ylabel="")
putil.plot_conditional_median_quantiles(axarr[1], samples, pred_names,
                                        data['X'][:,1], -9, 9, 1,
                                        data['y'], residual=True,
                                        xlabel="x2", ylabel="")
putil.plot_conditional_median_quantiles(axarr[2], samples, pred_names,
                                        data['X'][:,2], -9, 9, 1,
                                        data['y'], residual=True,
                                        xlabel="x3", ylabel="")
plot.show()
```

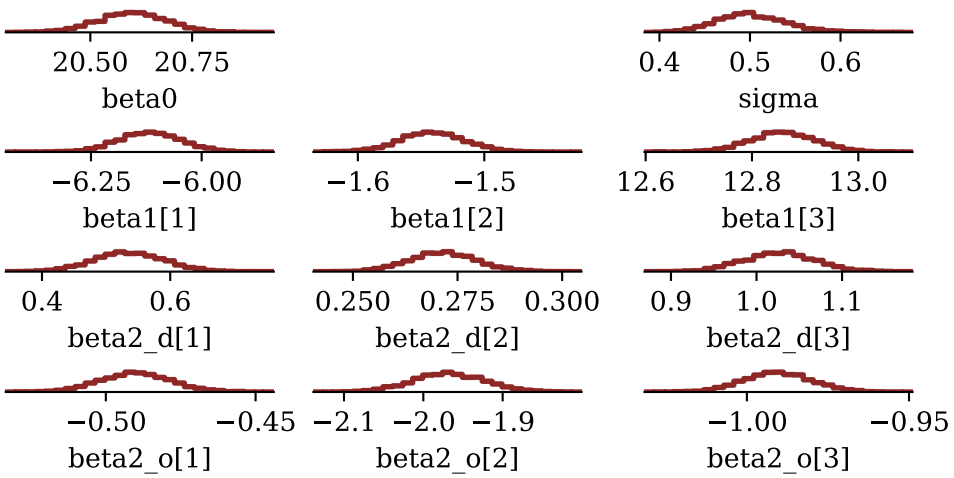




With no indications of model inadequacy we can move onto our posterior inferences. As before we can visualize the pushforward posterior distributions for each individual, one-dimensional parameter.

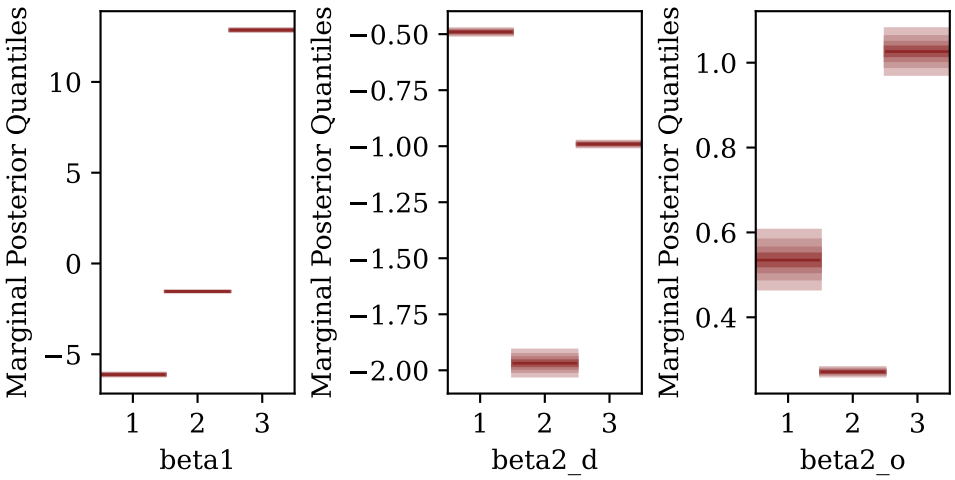
```
f, axarr = plot.subplots(4, 3, layout="constrained")

util.plot_expectand_pushforward(axarr[0, 0], samples['beta0'],
                                25, display_name="beta0")

axarr[0, 1].axis('off')

util.plot_expectand_pushforward(axarr[0, 2], samples['sigma'],
                                25, display_name="sigma")

for m in range(data['M']):
  name = f'beta1[{m + 1}]'
  util.plot_expectand_pushforward(axarr[1, m], samples[name],
                                  25, display_name=name)

for m in range(data['M']):
  name = f'beta2_d[{m + 1}]'
  util.plot_expectand_pushforward(axarr[2, m], samples[name],
                                  25, display_name=name)

for m in range(data['M']):
  name = f'beta2_o[{m + 1}]'
  util.plot_expectand_pushforward(axarr[3, m], samples[name],
                                  25, display_name=name)
```

```
plot.show()
```

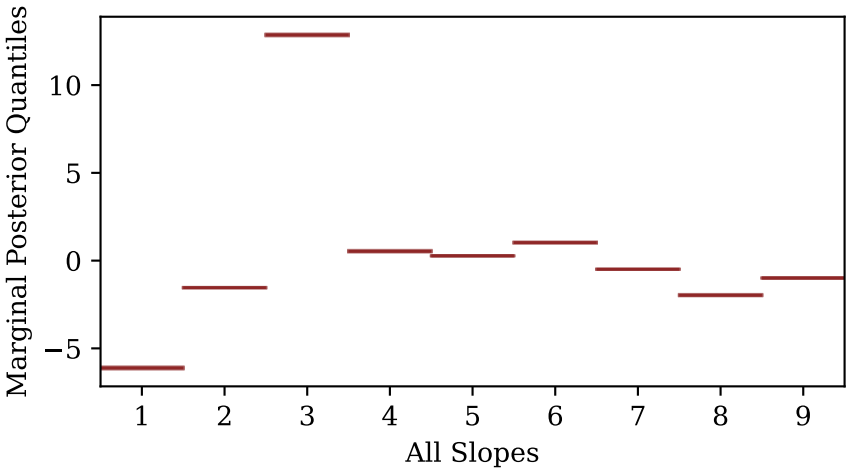


The `plot_disc_pushforward_quantiles` function plots disconnected, marginal nested quantile intervals for a collection of one-dimensional variables. This allows for a more compact visualization of the marginal posterior distributions.

```
f, axarr = plot.subplots(1, 3, layout="constrained")

names = [ f'beta1[{m + 1}]' for m in range(data['M']) ]
putil.plot_disc_pushforward_quantiles(axarr[0], samples, names,
                                      xlabel="beta1",
                                      ylabel="Marginal Posterior Quantiles")

names = [ f'beta2_o[{m + 1}]' for m in range(data['M']) ]
putil.plot_disc_pushforward_quantiles(axarr[1], samples, names,
                                      xlabel="beta2_d",
                                      ylabel="Marginal Posterior Quantiles")

names = [ f'beta2_d[{m + 1}]' for m in range(data['M']) ]
putil.plot_disc_pushforward_quantiles(axarr[2], samples, names,
                                      xlabel="beta2_o",
                                      ylabel="Marginal Posterior Quantiles")

plot.show()
```

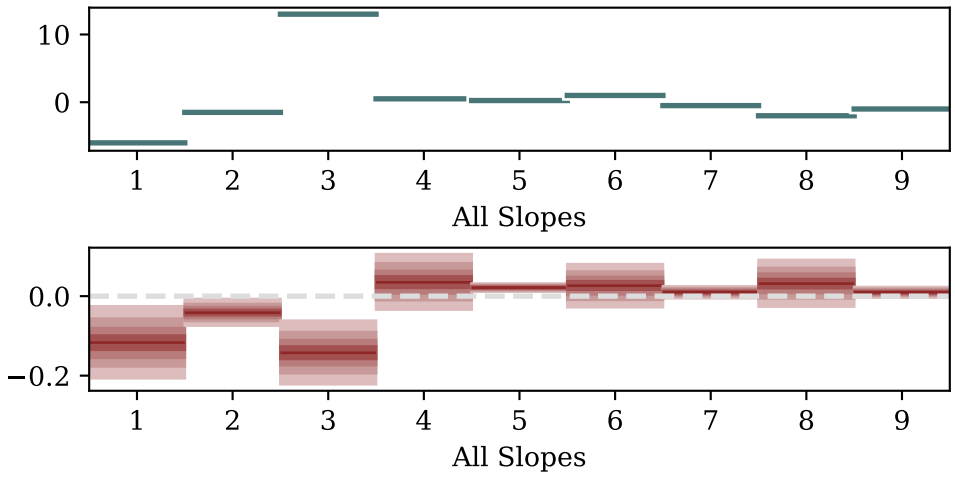



```
names = [ f'beta1[{m + 1}]' for m in range(data['M']) ] + \
        [ f'beta2_d[{m + 1}]' for m in range(data['M']) ] + \
        [ f'beta2_o[{m + 1}]' for m in range(data['M']) ]
putil.plot_disc_pushforward_quantiles(plot.gca(), samples, names,
                                      xlabel="All Slopes",
                                      ylabel="Marginal Posterior Quantiles")
plot.show()
```



This function also includes an optional `baseline_values` argument and `residual` configuration which we can use to compare the probabilistic to the point values, for example our marginal posterior inferences to the true values when analyzing simulated data.

```
true_slopes = [-6.00, -1.50, 13.00,  0.50, 0.25,
               1.00, -0.50, -2.00, -1.00]

f, axarr = plot.subplots(2, 1, layout="constrained")

putil.plot_disc_pushforward_quantiles(axarr[0], samples, names,
                                      baseline_values=true_slopes,
                                      baseline_color=putil.mid_teal,
                                      xlabel="All Slopes", ylabel="")

putil.plot_disc_pushforward_quantiles(axarr[1], samples, names,
                                      baseline_values=true_slopes,
                                      residual=True,
                                      xlabel="All Slopes", ylabel="")

plot.show()
```



License

https://creativecommons.org/licenses/by-nc/4.0/

Original Computing Environment

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

```
Last updated: 2024-07-16T18:10:54.253437-04:00

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, json, stan"))
```

```
matplotlib: 3.8.0
 numpy    : not installed
 json     : not installed
 stan     : not installed
```

Stan
Program 2 `uni\_full\_model.stan`

```
data {
  int<lower=1> N;
  vector[N] x; // Observed inputs
  vector[N] y; // Observed outputs

  int<lower=1> N_grid; // Number of grid points for quantifying functional behavior
  vector[N_grid] x_grid; // Grid points for quantifying functional behavior
}

parameters {
  real alpha;
  real beta;
  real<lower=0> sigma;
}

model {
  alpha ~ normal(0, 3 / 2.32);
  beta ~ normal(0, 3 / 2.32);
  sigma ~ normal(0, 1 / 2.57);

  y ~ normal(alpha + beta * x, sigma);
}

generated quantities {
  vector[N_grid] f_grid = alpha + beta * x_grid;
  array[N_grid] real y_pred_grid = normal_rng(f_grid, sigma);
}
```

Stan
Program 3 `multi\_prior\_model.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
}

transformed data {
  matrix[N, M * (M + 3) / 2 + 1] deltaX;
  for (n in 1:N) {
    deltaX[n, 1] = 1;

    for (m1 in 1:M) {
      // Linear perturbations
      deltaX[n, m1 + 1] = X[n, m1] - x0[m1];
    }

    for (m1 in 1:M) {
      // On-diagonal quadratic perturbations
      deltaX[n, M + m1 + 1]
        = deltaX[n, m1 + 1] * deltaX[n, m1 + 1];

      for (m2 in (m1 + 1):M) {
        int m3 = (2 * M - m1) * (m1 - 1) / 2 + m2 - m1;

        // Off-diagonal quadratic perturbations
        // Factor of 2 ensures that beta parameters have the
        // same interpretation as the expanded implementation
        deltaX[n, 2 * M + m3 + 1]
          = 2 * deltaX[n, m1 + 1] * deltaX[n, m2 + 1];
      }
    }
  }
}

parameters {
  real beta0;                      // Intercept
  vector[M] beta1;                 // Linear slopes
  vector[M] beta2_d;               // On-diagonal quadratic slopes
  vector[M * (M - 1) / 2] beta2_o; // Off-diagonal quadratic slopes
  real<lower=0> sigma;             // Measurement Variability
}

model {
  vector[M * (M + 3) / 2 + 1] beta
    = append_row(
        append_row(
          append_row(beta0, beta1),
        beta2_d),
```

Stan
Program 4 `multi\_full\_model.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] real y; // Variates
}

transformed data {
  matrix[N, M * (M + 3) / 2 + 1] deltaX;
  for (n in 1:N) {
    deltaX[n, 1] = 1;

    for (m1 in 1:M) {
      // Linear perturbations
      deltaX[n, m1 + 1] = X[n, m1] - x0[m1];
    }

    for (m1 in 1:M) {
      // On-diagonal quadratic perturbations
      deltaX[n, M + m1 + 1]
        = deltaX[n, m1 + 1] * deltaX[n, m1 + 1];

      for (m2 in (m1 + 1):M) {
        int m3 = (2 * M - m1) * (m1 - 1) / 2 + m2 - m1;

        // Off-diagonal quadratic perturbations
        // Factor of 2 ensures that beta parameters have the
        // same interpretation as the expanded implementation
        deltaX[n, 2 * M + m3 + 1]
          = 2 * deltaX[n, m1 + 1] * deltaX[n, m2 + 1];
      }
    }
  }
}

parameters {
  real beta0;                     // Intercept
  vector[M] beta1;                // Linear slopes
  vector[M] beta2_d;              // On-diagonal quadratic slopes
  vector[M * (M - 1) / 2] beta2_o; // Off-diagonal quadratic slopes
  real<lower=0> sigma;            // Measurement Variability
}

model {
  vector[M * (M + 3) / 2 + 1] beta
    = append_row(
        append_row(
```