

Monotonic Regression

Michael Betancourt

October 2026

Table of contents

1 Environment Setup	1
2 Implementing M-Splines and I-Splines	2
3 Explore Data	3
4 Prior Checks	5
5 Posterior Inferences	13
References	17
License	17
Original Computing Environment	17

In this short note I review the implementation of M-splines and I-splines in **Stan**, and then apply the latter to a basic monotonic analysis.

1 Environment Setup

We begin by configuring our local R environment.

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

```
nom_colors <- c("#DCBCBC", "#C79999", "#B97C7C",  
               "#A25050", "#8F2727", "#7C0000")
```

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

To facilitate the implementation of the Bayesian analysis we'll take advantage of my recommended diagnostic and visualization tools.

```
util <- new.env()
```

First we have a suite of Markov chain Monte Carlo diagnostics and estimation tools; this code and supporting documentation are both available on [GitHub](#).

```
source('mcmc_analysis_tools_rstan.R', local=util)
```

Second we have a suite of probabilistic visualization functions based on Markov chain Monte Carlo estimation. Again the code and supporting documentation are available on [GitHub](#).

```
source('mcmc_visualization_tools.R', local=util)
```

2 Implementing M-Splines and I-Splines

M-splines are non-negative piecewise polynomials. Integrating M-splines defines monotonically non-decreasing piecewise polynomials known as I-splines. The “I” in “I-spline” is reasonable, but the “M” in “M-spline” is a bit more baffling as M-splines are not themselves monotonic.

In typical spline fashion, M-splines and I-splines are typically implemented as weighted sums of basis functions,

$$M(x; \mathbf{t}) = \sum_{i=1}^I w_i M_i(x; k, \mathbf{t})$$
$$I(x; \mathbf{t}) = \sum_{i=1}^I w_i I_i(x; k, \mathbf{t}).$$

Here k is the order of spline basis and $\mathbf{t}$ are knots defined by a grid of input values along with repeated boundaries.

The classic reference for M-splines and I-splines is Ramsay (1988). This review details how M-spline basis functions of a given order can be derived recursively from lower-order basis functions, and how I-spline basis functions of a given order can be derived from higher-order M-spline basis functions. Unfortunately, the I-spline equation is incorrect. The correct equation can be found [here](#) and [here](#).

Often, the most convenient way to implement M-splines or I-splines in practice is to evaluate the basis functions to the desired order at each input of interest and then organize those evaluations into a matrix. Then we can efficiently evaluate weighted sums of basis functions with a matrix-vector multiplication.

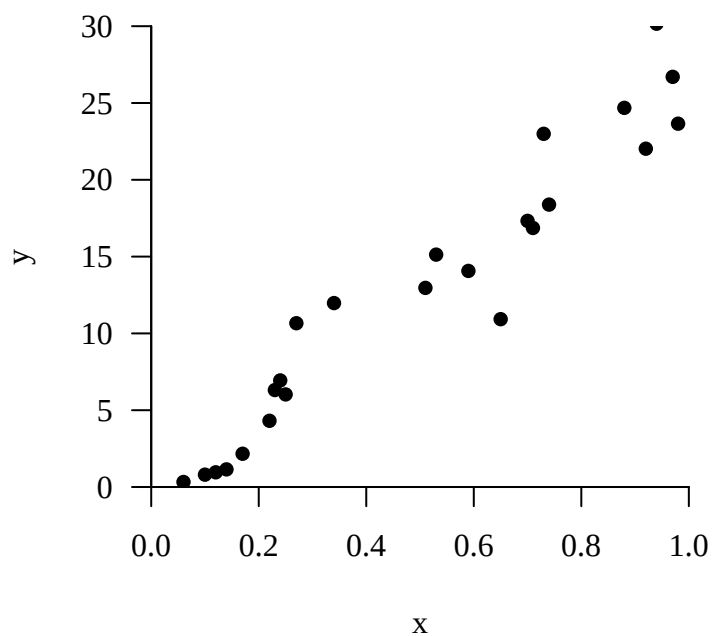
3 Explore Data

Once we can construct I-splines, we can use them to model a monotonically non-decreasing location function in a one-dimensional regression model. Here we'll construct such a model to analyze the following observations.

```
data <- read_rdump('data/obs.R')

par(mfrow=c(1, 1))

plot(data$obs_xs, data$obs_ys,
      cex=1, pch=16, col='black',
      xlim=c(0, 1), xlab='x',
      ylim=c(0, 30), ylab='y')
```



```
data.frame(data$obs_xs, data$obs_ys)
```

	data.obs_xs	data.obs_ys
1	0.06	0.3264292
2	0.10	0.8038582
3	0.12	0.9565876
4	0.14	1.1472394
5	0.17	2.1641812
6	0.22	4.3103346
7	0.23	6.3129131
8	0.24	6.9385851
9	0.25	6.0286320
10	0.27	10.6626544
11	0.34	11.9748162
12	0.51	12.9632037
13	0.53	15.1286951
14	0.59	14.0670644
15	0.65	10.9262495
16	0.70	17.3263034
17	0.71	16.8643068
18	0.73	22.9923982
19	0.74	18.3862439
20	0.86	36.0855580
21	0.88	24.6834362

22	0.92	22.0244512
23	0.94	30.1607568
24	0.97	26.7039537
25	0.98	23.6534994

4 Prior Checks

Given a spline order and a discretization of the covariate space, a spline model is configured by the choice of basis function weights. Engineering a prior model over the weights that results in reasonably functional behavior, however, requires care.

Let's say, for example, that our domain expertise is consistent with the fixed, lower boundary condition $f(x_{\min}) = 0$ and the more uncertain, upper boundary condition

$$0 \leq f(x_{\max}) \lesssim 50.$$

We can enforce the lower boundary condition by fixing the first weight to be zero. The behavior at the upper boundary, however, depends on the behavior of all of the intermediate weights. What kind of prior model ensures that behavior?

Let's start with an independent normal prior model on all of the free weights. Here I've tuned the scale of the component prior models to achieve the desired behavior, with the intuition that each I-spline basis function maxes out at one.

```
data$xis <- seq(0, 1, 0.1)
data$N <- length(data$xis)

data$x_grid <- seq(0, 1 - 0.01, 0.01)
data$N_grid <- length(data$x_grid)
```

```
prior_samples <- list()
```

```
prior <- stan(file="stan_programs/half_normal_prior.stan",
             data=data, seed=8438338,
             warmup=1000, iter=2024, refresh=0)

prior_samples[[1]] <- util$extract_expectand_vals(prior)
```

The immediate issue with this prior model is that it strongly prefers relatively uniform weight configurations. This then suppresses many reasonable functional behaviors, such as those that grow especially quickly or especially slowly.

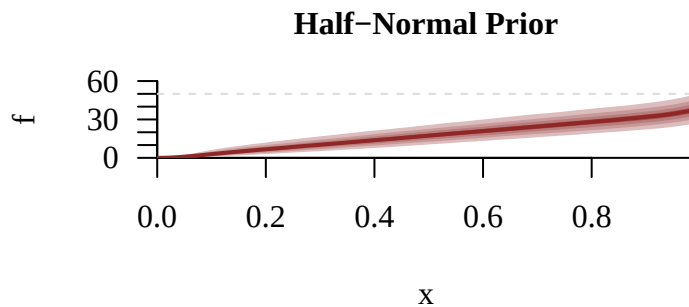
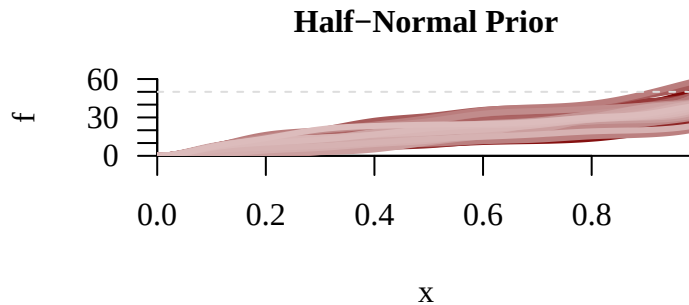
```
f_names <- sapply(1:data$N_grid,
                 function(n) paste0('fs[', n, ']'))

s <- 1
prior_name <- 'Half-Normal'

par(mfrow=c(2, 1))

util$plot_realizations(prior_samples[[s]], f_names, data$x_grid,
                      xlab="x",
                      ylab="f", display_ylim=c(0, 60),
                      main=paste(prior_name, 'Prior'))
abline(h=50, lwd=1, lty=2, col="#DDDDDD")

util$plot_conn_pushforward_quantiles(
  prior_samples[[s]], f_names, data$x_grid,
  xlab="x",
  ylab="f", display_ylim=c(0, 60),
  main=paste(prior_name, 'Prior')
)
abline(h=50, lwd=1, lty=2, col="#DDDDDD")
```



One way to allow for less rigid functional behaviors is to use a more heavy-tailed prior model.

```
prior <- stan(file="stan_programs/half_cauchy_prior.stan",
             data=data, seed=8438338,
             warmup=1000, iter=2024, refresh=0)

prior_samples[[2]] <- util$extract_expectand_vals(prior)
```

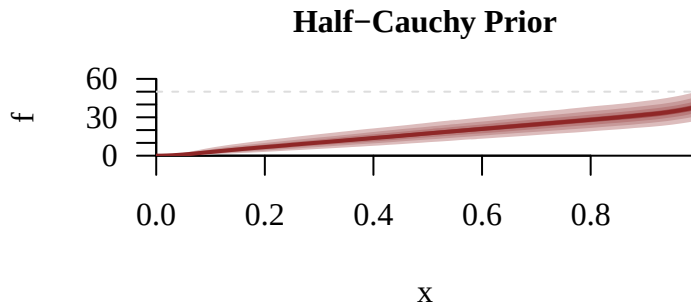
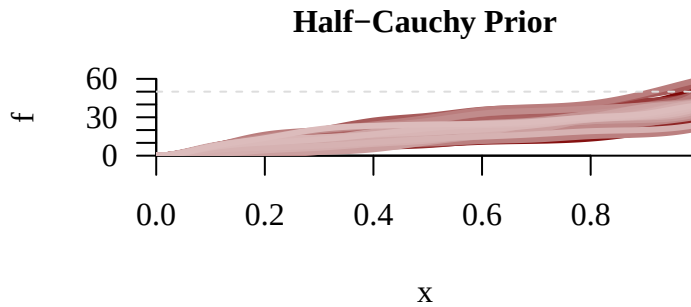
Indeed, we see a much wider range of functional behaviors.

```
s <- 1
prior_name <- 'Half-Cauchy'

par(mfrow=c(2, 1))

util$plot_realizations(prior_samples[[s]], f_names, data$x_grid,
                      xlab="x",
                      ylab="f", display_ylim=c(0, 60),
                      main=paste(prior_name, 'Prior'))
abline(h=50, lwd=1, lty=2, col="#DDDDDD")

util$plot_conn_pushforward_quantiles(
  prior_samples[[s]], f_names, data$x_grid,
  xlab="x",
  ylab="f", display_ylim=c(0, 60),
  main=paste(prior_name, 'Prior')
)
abline(h=50, lwd=1, lty=2, col="#DDDDDD")
```



An immediate issue with an independent Cauchy model, however, is that we have no way of tuning exactly how sparse the weights should be. A mixture of normals, with one narrow component and one wide component, offers a bit more interpretable flexibility.

```
prior <- stan(file="stan_programs/mix_half_normal_prior.stan",
              data=data, seed=8438338,
              warmup=1000, iter=2024, refresh=0)

prior_samples[[3]] <- util$extract_expectand_vals(prior)
```

That said, the functional behaviors don't end up being much different from what we saw with the independent normal prior model. We would probably have to play around with the scales to allow for more diverse possibilities.

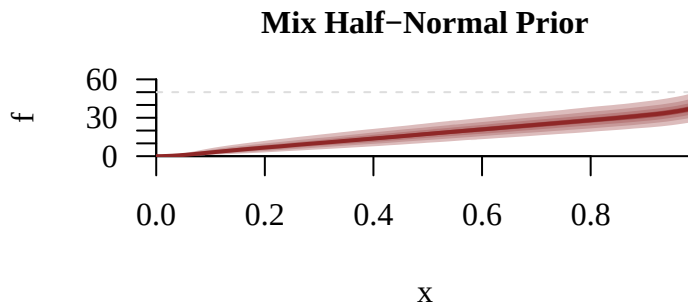
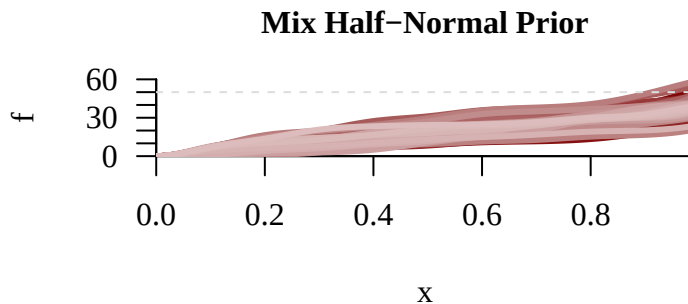
```
s <- 1
prior_name <- 'Mix Half-Normal'

par(mfrow=c(2, 1))

util$plot_realizations(prior_samples[[s]], f_names, data$x_grid,
                      xlab="x",
                      ylab="f", display_ylim=c(0, 60),
                      main=paste(prior_name, 'Prior'))
```

```
abline(h=50, lwd=1, lty=2, col="#DDDDDD")

util$plot_conn_pushforward_quantiles(
  prior_samples[[s]], f_names, data$x_grid,
  xlab="x",
  ylab="f", display_ylim=c(0, 60),
  main=paste(prior_name, 'Prior')
)
abline(h=50, lwd=1, lty=2, col="#DDDDDD")
```



Another interesting possibility is a scaled Dirichlet model. Here we derive the free weights from a simplex configuration multiplied by a positive scale. This decouples the weight behavior into relative basis function contributions and a global magnitude.

For example, if we want to allow more flexible functional behaviors then we would need more diversity across the weights. In a scaled Dirichlet model we can ensure this by favoring simplex configurations closer to the simplex boundary.

```
prior <- stan(file="stan_programs/scaled_dirichlet_prior.stan",
  data=data, seed=8438338,
  warmup=1000, iter=2024, refresh=0)

prior_samples[[4]] <- util$extract_expectand_vals(prior)
```

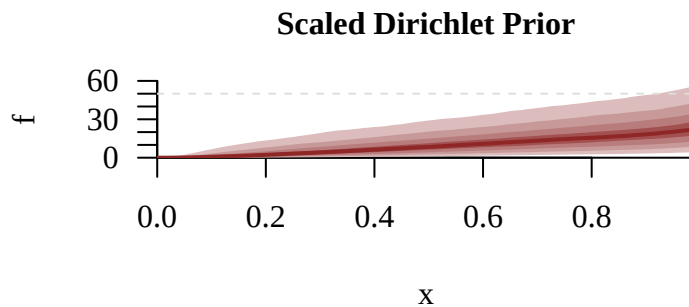
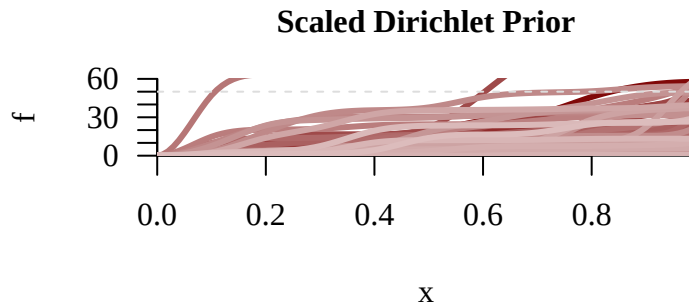
The prior functional behaviors are similar to what we saw with the independent Cauchy prior model, although a bit more symmetric.

```
s <- 4
prior_name <- 'Scaled Dirichlet'

par(mfrow=c(2, 1))

util$plot_realizations(prior_samples[[s]], f_names, data$x_grid,
                      xlab="x",
                      ylab="f", display_ylim=c(0, 60),
                      main=paste(prior_name, 'Prior'))
abline(h=50, lwd=1, lty=2, col="#DDDDDD")

util$plot_conn_pushforward_quantiles(
  prior_samples[[s]], f_names, data$x_grid,
  xlab="x",
  ylab="f", display_ylim=c(0, 60),
  main=paste(prior_name, 'Prior')
)
abline(h=50, lwd=1, lty=2, col="#DDDDDD")
```



For a more direct comparison, we can plot all four functional side-by-side.

```

prior_names <- c('Half-Normal', 'Half-Cauchy',
                 'Mix Half-Normal', 'Scaled Dirichlet')

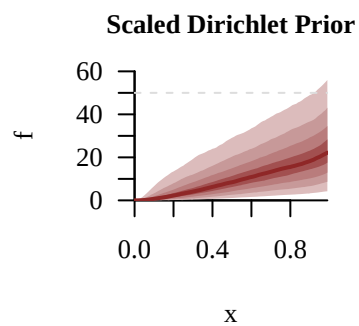
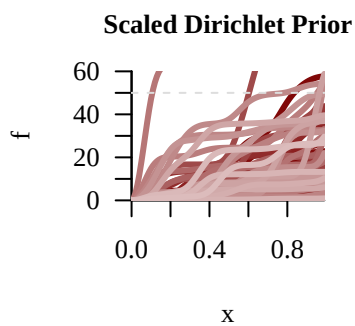
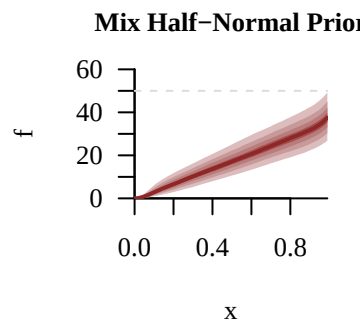
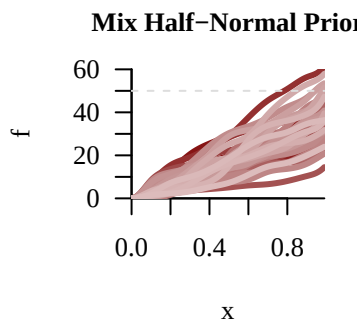
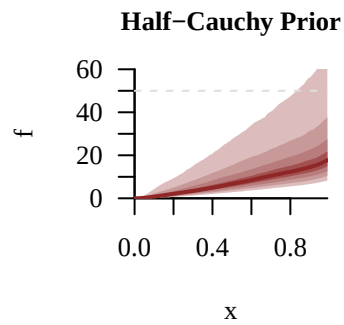
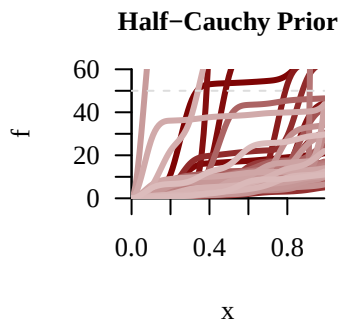
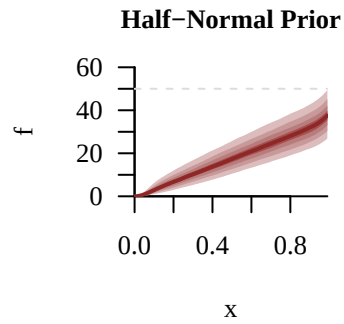
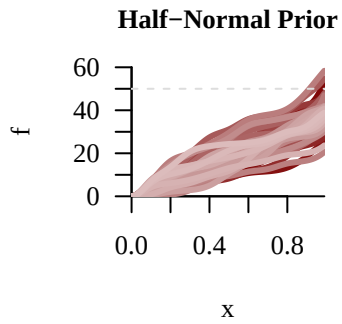
f_names <- sapply(1:data$N_grid,
                 function(n) paste0('fs[', n, ']''))

par(mfrow=c(2, 2))

for (s in 1:4) {
  util$plot_realizations(prior_samples[[s]],
                        f_names, data$x_grid,
                        xlab="x",
                        ylab="f", display_ylim=c(0, 60),
                        main=paste(prior_names[s], 'Prior'))
  abline(h=50, lwd=1, lty=2, col="#DDDDDD")

  util$plot_conn_pushforward_quantiles(
    prior_samples[[s]], f_names, data$x_grid,
    xlab="x",
    ylab="f", display_ylim=c(0, 60),
    main=paste(prior_names[s], 'Prior')
  )
  abline(h=50, lwd=1, lty=2, col="#DDDDDD")
}

```



These comparisons just scratch the surface of principled spline prior models. That said, for this demonstration let's move forward using the scaled Dirichlet prior model.

5 Posterior Inferences

Given the I-spline model, implementing a monotonic regression model is straightforward.

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

The computational diagnostics show only some indications of heavy-tails in the posterior distribution, which will not be problematic for our inferences.

```
diagnostics <- util$extract_hmc_diagnostics(fit)
util$check_all_hmc_diagnostics(diagnostics)
```

Chain 4: 1 of 1024 transitions (0.1%) 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 `adapt_delta` larger than 0.801 may reduce the instabilities at the cost of more expensive Hamiltonian transitions.

```
samples <- util$extract_expectand_vals(fit)
base_samples <- util$filter_expectands(samples,
                                       c('rho', 'alpha', 'sigma'),
                                       check_arrays=TRUE)
util$check_all_expectand_diagnostics(base_samples)
```

rho[5]:

Chain 1: Right tail $\hat{\text{rho}}$ (0.266) exceeds 0.25.

Chain 4: Right tail $\hat{\text{rho}}$ (0.307) exceeds 0.25.

rho[6]:

Chain 1: Right tail $\hat{\text{rho}}$ (0.261) exceeds 0.25.

Chain 2: Right tail $\hat{\text{rho}}$ (0.277) exceeds 0.25.

rho[9]:

Chain 1: Right tail $\hat{\text{rho}}$ (0.307) exceeds 0.25.

Chain 2: Right tail $\hat{\text{rho}}$ (0.282) exceeds 0.25.

Chain 3: Right tail $\hat{\xi}$ (0.450) exceeds 0.25.
Chain 4: Right tail $\hat{\xi}$ (0.372) exceeds 0.25.

rho[10]:

Chain 1: Right tail $\hat{\xi}$ (0.399) exceeds 0.25.
Chain 2: Right tail $\hat{\xi}$ (0.352) exceeds 0.25.
Chain 3: Right tail $\hat{\xi}$ (0.346) exceeds 0.25.
Chain 4: Right tail $\hat{\xi}$ (0.301) exceeds 0.25.

rho[11]:

Chain 1: Right tail $\hat{\xi}$ (0.269) exceeds 0.25.
Chain 3: Right tail $\hat{\xi}$ (0.342) exceeds 0.25.
Chain 4: Right tail $\hat{\xi}$ (0.301) exceeds 0.25.

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

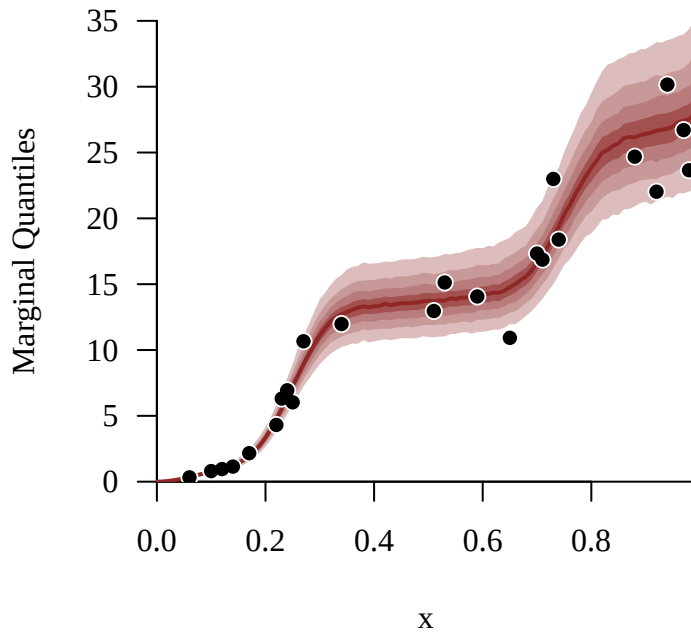
The lack of retrodictive tensions suggests that our monotonic functional model is sufficiently flexible for this data.

```
pred_names <- sapply(1:data$N_grid,
                    function(n) paste0('y_pred[', n, ']'))

par(mfrow=c(1, 1))

util$plot_conn_pushforward_quantiles(
  samples, pred_names, data$x_grid,
  xlab="x", display_ylim=c(0, 35)
)

points(data$obs_xs, data$obs_ys, cex=1.25, pch=16, col='white')
points(data$obs_xs, data$obs_ys, cex=1.00, pch=16, col='black')
```



Content with the adequacy of our model, we can investigate our posterior inferences. For this demonstrative analysis, I will compare the inferred functional behaviors to the true behavior from which the data were simulated. In this case, we have been able to accurately recover the true behavior.

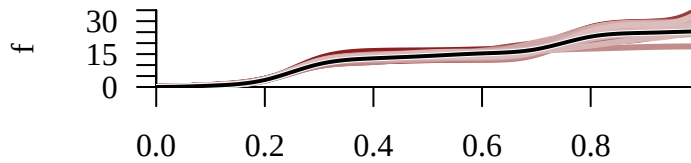
```
truth <- read_rdump('data/truth.R')

f_names <- sapply(1:data$N_grid,
                 function(n) paste0('fs[', n, ']'))

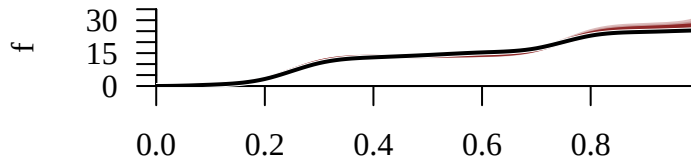
par(mfrow=c(2, 1))

util$plot_realizations(samples, f_names, data$x_grid,
                      xlab="x",
                      ylab="f", display_ylim=c(0, 35))
lines(truth$x_grid, truth$true_fs, lwd=3, col='white')
lines(truth$x_grid, truth$true_fs, lwd=2, col='black')

util$plot_conn_pushforward_quantiles(
  samples, f_names, data$x_grid,
  xlab="x", ylab="f", display_ylim=c(0, 35)
)
lines(truth$x_grid, truth$true_fs, lwd=3, col='white')
lines(truth$x_grid, truth$true_fs, lwd=2, col='black')
```



x



x

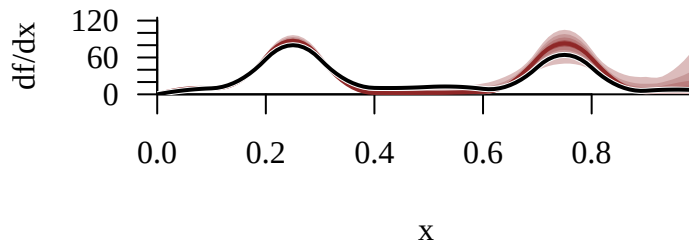
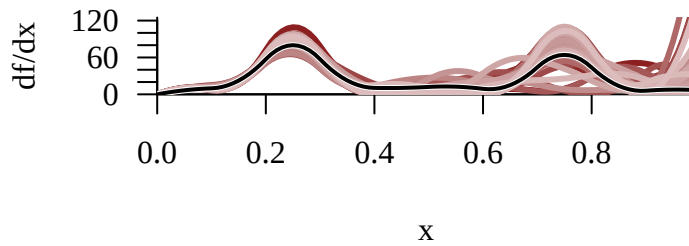
A nice feature of the I-spline construction is we can readily recover differential information from the M-spline basis functions. This is especially useful when the non-negative derivative function is more relevant to the system being studied.

```
f_names <- sapply(1:data$N_grid,
                  function(n) paste0('dfdxs[' , n, ']'))

par(mfrow=c(2, 1))

util$plot_realizations(samples, f_names, data$x_grid,
                      xlab="x", ylab="df/dx", display_ylim=c(0, 125))
lines(truth$x_grid, truth$true_dfdxs, lwd=3, col='white')
lines(truth$x_grid, truth$true_dfdxs, lwd=2, col='black')

util$plot_conn_pushforward_quantiles(
  samples, f_names, data$x_grid,
  xlab="x", ylab="df/dx", display_ylim=c(0, 125)
)
lines(truth$x_grid, truth$true_dfdxs, lwd=3, col='white')
lines(truth$x_grid, truth$true_dfdxs, lwd=2, col='black')
```



References

Ramsay, J. O. 1988. “Monotone Regression Splines in Action.” *Statistical Science* 3 (4): 425–41.

License

The code in this case study is copyrighted by Michael Betancourt and licensed under the new BSD (3-clause) license:

<https://opensource.org/licenses/BSD-3-Clause>

The text and figures in this chapter are copyrighted by Michael Betancourt and licensed under the CC BY-NC 4.0 license:

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

Original Computing Environment

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

CC=clang

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

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

sessionInfo()

R version 4.3.2 (2023-10-31)
Platform: x86_64-apple-darwin20 (64-bit)
Running under: macOS 15.7.4

Matrix products: default
BLAS: /Library/Frameworks/R.framework/Versions/4.3-x86_64/Resources/lib/libRblas.0.dylib
LAPACK: /Library/Frameworks/R.framework/Versions/4.3-x86_64/Resources/lib/libRlapack.dylib;

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

time zone: America/New_York
tzcode source: internal

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

other attached packages:
[1] rstan_2.32.6 StanHeaders_2.32.7 colormap_0.1.4

loaded via a namespace (and not attached):
[1] gtable_0.3.4 jsonlite_1.8.8 compiler_4.3.2 Rcpp_1.0.11
[5] stringr_1.5.1 parallel_4.3.2 gridExtra_2.3 scales_1.3.0
[9] yaml_2.3.8 fastmap_1.1.1 ggplot2_3.4.4 R6_2.6.1
[13] curl_5.2.0 knitr_1.45 tibble_3.2.1 munsell_0.5.0
[17] pillar_1.9.0 rlang_1.1.2 utf8_1.2.4 V8_4.4.1
[21] stringi_1.8.3 inline_0.3.19 xfun_0.41 RcppParallel_5.1.7
[25] cli_3.6.2 magrittr_2.0.3 digest_0.6.33 grid_4.3.2
[29] lifecycle_1.0.4 vctrs_0.6.5 evaluate_0.23 glue_1.6.2
[33] QuickJSR_1.0.8 codetools_0.2-19 stats4_4.3.2 pkgbuild_1.4.3
[37] fansi_1.0.6 colorspace_2.1-0 rmarkdown_2.25 matrixStats_1.2.0
[41] tools_4.3.2 loo_2.6.0 pkgconfig_2.0.3 htmltools_0.5.7

```

functions {
  int n_bases(int k, vector xis) {
    return size(xis) + k - 2;
  }

  matrix eval_M_bases(vector xs, int k, vector xis) {
    int S = size(xs);
    int N = size(xis);

    int I = N + k - 2;
    int L = N + 2 * (k - 1);

    vector[L] taus = rep_vector(not_a_number(), L);
    matrix[S, I] M;

    int II = N + 1 - 2;
    int LL = N + 2 * (1 - 1);

    if (min(xs) < xis[1])
      reject("Evaluation points must bounded ",
            "below by the first grid point." );

    if (max(xs) > xis[N])
      reject("Evaluation points must bounded ",
            "above by the last grid point." );

    if (k < 1)
      reject("Order must be positive.");

    // First-order initialization
    taus[1:N] = xis;

    for (i in 1:II)
      for (s in 1:S)
        M[s, i] = taus[i] <= xs[s] && xs[s] < taus[i + 1] ?
                  1 / (taus[i + 1] - taus[i]) : 0;

    if (k == 1) return M;

    // Recursive order updates
    for (kk in 2:k) {
      real C;

      taus[1:(LL + 2)]
        = append_row(taus[1:1],
                     append_row(taus[1:LL], taus[LL:LL]));

      II = N + kk - 2;
      LL = N + 2 * (kk - 1);
    }
  }
}

```

Stan

Program 2 half_normal_prior.stan

```
functions {
  int n_bases(int k, vector xis) {
    return size(xis) + k - 2;
  }

  matrix eval_M_bases(vector xs, int k, vector xis) {
    int S = size(xs);
    int N = size(xis);

    int I = N + k - 2;
    int L = N + 2 * (k - 1);

    vector[L] taus = rep_vector(not_a_number(), L);
    matrix[S, I] M;

    int II = N + 1 - 2;
    int LL = N + 2 * (1 - 1);

    if (min(xs) < xis[1])
      reject("Evaluation points must bounded ",
            "below by the first grid point." );

    if (max(xs) > xis[N])
      reject("Evaluation points must bounded ",
            "above by the last grid point." );

    if (k < 1)
      reject("Order must be positive.");

    // First-order initialization
    taus[1:N] = xis;

    for (i in 1:II)
      for (s in 1:S)
        M[s, i] = taus[i] <= xs[s] && xs[s] < taus[i + 1] ?
                  1 / (taus[i + 1] - taus[i]) : 0;

    if (k == 1) return M;

    // Recursive order updates
    for (kk in 2:k) {
      real C;

      taus[1:(LL + 2)]
        = append_row(taus[1:1],
                     append_row(taus[1:LL], taus[LL:LL]));

      II = N + kk - 2;
      LL = N + 2 * (kk - 1);
    }
  }
}
```

Stan

Program 3 half_cauchy_prior.stan

```
functions {
  int n_bases(int k, vector xis) {
    return size(xis) + k - 2;
  }

  matrix eval_M_bases(vector xs, int k, vector xis) {
    int S = size(xs);
    int N = size(xis);

    int I = N + k - 2;
    int L = N + 2 * (k - 1);

    vector[L] taus = rep_vector(not_a_number(), L);
    matrix[S, I] M;

    int II = N + 1 - 2;
    int LL = N + 2 * (1 - 1);

    if (min(xs) < xis[1])
      reject("Evaluation points must bounded ",
            "below by the first grid point." );

    if (max(xs) > xis[N])
      reject("Evaluation points must bounded ",
            "above by the last grid point." );

    if (k < 1)
      reject("Order must be positive.");

    // First-order initialization
    taus[1:N] = xis;

    for (i in 1:II)
      for (s in 1:S)
        M[s, i] = taus[i] <= xs[s] && xs[s] < taus[i + 1] ?
          1 / (taus[i + 1] - taus[i]) : 0;

    if (k == 1) return M;

    // Recursive order updates
    for (kk in 2:k) {
      real C;

      taus[1:(LL + 2)]
        = append_row(taus[1:1],
                     append_row(taus[1:LL], taus[LL:LL]));

      II = N + kk - 2;
      LL = N + 2 * (kk - 1);
    }
  }
}
```

```

functions {
  int n_bases(int k, vector xis) {
    return size(xis) + k - 2;
  }

  matrix eval_M_bases(vector xs, int k, vector xis) {
    int S = size(xs);
    int N = size(xis);

    int I = N + k - 2;
    int L = N + 2 * (k - 1);

    vector[L] taus = rep_vector(not_a_number(), L);
    matrix[S, I] M;

    int II = N + 1 - 2;
    int LL = N + 2 * (1 - 1);

    if (min(xs) < xis[1])
      reject("Evaluation points must bounded ",
            "below by the first grid point." );

    if (max(xs) > xis[N])
      reject("Evaluation points must bounded ",
            "above by the last grid point." );

    if (k < 1)
      reject("Order must be positive.");

    // First-order initialization
    taus[1:N] = xis;

    for (i in 1:II)
      for (s in 1:S)
        M[s, i] = taus[i] <= xs[s] && xs[s] < taus[i + 1] ?
                  1 / (taus[i + 1] - taus[i]) : 0;

    if (k == 1) return M;

    // Recursive order updates
    for (kk in 2:k) {
      real C;

      taus[1:(LL + 2)]
        = append_row(taus[1:1],
                     append_row(taus[1:LL], taus[LL:LL]));

      II = N + kk - 2;
      LL = N + 2 * (kk - 1);
    }
  }
}

```

```

functions {
  int n_bases(int k, vector xis) {
    return size(xis) + k - 2;
  }

  matrix eval_M_bases(vector xs, int k, vector xis) {
    int S = size(xs);
    int N = size(xis);

    int I = N + k - 2;
    int L = N + 2 * (k - 1);

    vector[L] taus = rep_vector(not_a_number(), L);
    matrix[S, I] M;

    int II = N + 1 - 2;
    int LL = N + 2 * (1 - 1);

    if (min(xs) < xis[1])
      reject("Evaluation points must bounded ",
            "below by the first grid point." );

    if (max(xs) > xis[N])
      reject("Evaluation points must bounded ",
            "above by the last grid point." );

    if (k < 1)
      reject("Order must be positive.");

    // First-order initialization
    taus[1:N] = xis;

    for (i in 1:II)
      for (s in 1:S)
        M[s, i] = taus[i] <= xs[s] && xs[s] < taus[i + 1] ?
                  1 / (taus[i + 1] - taus[i]) : 0;

    if (k == 1) return M;

    // Recursive order updates
    for (kk in 2:k) {
      real C;

      taus[1:(LL + 2)]
        = append_row(taus[1:1],
                     append_row(taus[1:LL], taus[LL:LL]));

      II = N + kk - 2;
      LL = N + 2 * (kk - 1);
    }
  }
}

```

```

functions {
  int n_bases(int k, vector xis) {
    return size(xis) + k - 2;
  }

  matrix eval_M_bases(vector xs, int k, vector xis) {
    int S = size(xs);
    int N = size(xis);

    int I = N + k - 2;
    int L = N + 2 * (k - 1);

    vector[L] taus = rep_vector(not_a_number(), L);
    matrix[S, I] M;

    int II = N + 1 - 2;
    int LL = N + 2 * (1 - 1);

    if (min(xs) < xis[1])
      reject("Evaluation points must bounded ",
            "below by the first grid point." );

    if (max(xs) > xis[N])
      reject("Evaluation points must bounded ",
            "above by the last grid point." );

    if (k < 1)
      reject("Order must be positive.");

    // First-order initialization
    taus[1:N] = xis;

    for (i in 1:II)
      for (s in 1:S)
        M[s, i] = taus[i] <= xs[s] && xs[s] < taus[i + 1] ?
          1 / (taus[i + 1] - taus[i]) : 0;

    if (k == 1) return M;

    // Recursive order updates
    for (kk in 2:k) {
      real C;

      taus[1:(LL + 2)]
        = append_row(taus[1:1],
                     append_row(taus[1:LL], taus[LL:LL]));

      II = N + kk - 2;
      LL = N + 2 * (kk - 1);
    }
  }
}

```