BASS: An R Package for Fitting and Performing Sensitivity Analysis of Bayesian Adaptive Spline Surfaces

Devin Francom
University of California Santa Cruz

Bruno Sansó
University of California Santa Cruz

Abstract

We present the R package **BASS** as a tool for nonparametric regression. The primary focus of the package is fitting fully Bayesian adaptive spline surface (BASS) models and performing global sensitivity analyses of these models. The BASS framework is similar to that of Bayesian multivariate adaptive regression splines (BMARS) from Denison, Mallick, and Smith (1998), but with many added features. The software is built to efficiently handle significant amounts of data with many continuous or categorical predictors and with functional response. Under our Bayesian framework, most priors are automatic but these can be modified by the user to focus on parsimony and the avoidance of overfitting. If directed to do so, the software uses parallel tempering to improve the reversible jump Markov chain Monte Carlo (RJMCMC) methods used to perform inference. We discuss the implementation of these features and present the performance of **BASS** in a number of analyses of simulated and real data.

Keywords: splines, functional data analysis, sensitivity analysis, nonparametric regression.

1. Introduction

The purpose of the R (R Core Team 2016) package **BASS** (Francom 2016) is to provide an easy-to-use implementation of Bayesian adaptive spline models for nonparametric regression. It provides a combination of flexibility, scalability, interpretability and probabilistic accuracy that can be difficult to find in other nonparametric regression software packages. The model form is flexible enough to capture local features that may be present in the data. It is scalable to moderately large datasets in both the number of predictors and the number of observations. It performs automatic variable selection. It can build nonparametric functional regression models and incorporate categorical predictors. The package can partition the variability of a resultant model using a nonlinear ANOVA decomposition, providing valuable interpretation to the predictors. The Bayesian approach allows for model estimates and predictions that can be evaluated probabilistically. The package is available from the Comprehensive R Archive Network (CRAN) at `https://CRAN.R-project.org/package=BASS`.

The BASS framework builds on multivariate adaptive regression splines (MARS) from Friedman (1991b). A well-developed software implementation of the MARS model is available in the R package **earth** (Milborrow 2016). The Bayesian version of MARS (BMARS) was first developed in Denison *et al.* (1998). A MATLAB implementation of BMARS is available from

the software website accompanying Denison, Holmes, Mallick, and Smith (2002).

Our implementation is more similar to the BMARS implementation, though with some substantial changes to methodology as described in Francom, Sansó, Kupresanin, and Johannesson (2016). The primary motivation for developing this software was building surrogate models (or emulators) for complex and computationally expensive simulators (or computer models). In particular, we wanted to build a fast and accurate surrogate model to use for uncertainty quantification in the presence of a large number of simulations and where each simulation had functional output. Attributing the variance in the response of the surrogate to different combinations of predictors, a practice known as sensitivity analysis, is a valuable tool for determining which predictors and interactions are important. One of the major benefits of polynomial spline surrogate models is that sensitivity analysis can be done analytically. The **BASS** package has this functionality for scalar and functional response models.

We introduce the package as follows. In Section 2, we describe the modeling framework, including our methods for posterior sampling, modeling functional responses, and incorporating categorical inputs. In Section 3, we describe the sensitivity analysis methods. Then, in Section 4, we walk through six examples of how to use the package. These are done with **knitr** in order to be reproducible by the reader. Finally, in Section 5, present a summary of the package capabilities.

2. Bayesian Adaptive Spline Surfaces

The BASS model, like the MARS and BMARS models, uses data to learn a set of data dependent basis functions that are tensor products of polynomial splines. The number of basis functions as well as the knots and variables used in each basis are chosen adaptively. The BMARS approach uses reversible jump Markov chain Monte Carlo (RJMCMC) (Green 1995) to sample the posterior. The BASS adaptation of BMARS includes the improvements of Nott, Kuk, and Duc (2005) for more efficient posterior sampling as well as parallel tempering for better posterior exploration. BASS also efficiently handles functional responses and allows for categorical variables. We discuss each of these aspects below. First, we introduce the BASS model and priors.

Let y_i denote the dependent variable and $\mathbf{x}_i$ denote a vector of p independent variables, with $i = 1, \ldots, n$. Without loss of generality, let each independent variable be scaled to be between zero and one. We model y_i as

$$y_i = f(\mathbf{x}_i) + \epsilon_i, \quad \epsilon_i \sim N(0, \sigma^2) \tag{1}$$

$$f(\mathbf{x}) = a_0 + \sum_{m=1}^{M} a_m B_m(\mathbf{x}) \tag{2}$$

$$B_m(\mathbf{x}) = \prod_{k=1}^{K_m} g_{km}[s_{km}(x_{v_{km}} - t_{km})]_+^{\alpha} \tag{3}$$

where $s_{km} \in \{-1, 1\}$ is referred to as a sign, $t_{km} \in [0, 1]$ is a knot, v_{km} selects a variable, K_m is the degree of interaction and $g_{km} = (s_{km} + 1)/2 - s_{km} t_{km}$ is a constant that makes the basis function have a maximum of one. The function $[\cdot]_+$ is defined as $\max(0, \cdot)$. The power α determines the degree of the polynomial splines. We allow for no repeats in $v_{1m}, \ldots, v_{K_m m}$, meaning that a variable can be used only once in a basis function. M is the number of basis

functions, and $\mathbf{a}$ is the $M+1$ vector of basis coefficients (including the intercept). The only difference between this setup and that of MARS and BMARS is the inclusion of the constant g_{km} in each element of the tensor product. This normalizes the basis functions so that the basis coefficients $a_1, \ldots, a_M$ are on the same scale, making computations more stable.

In the course of fitting this model, we seek to estimate $\boldsymbol{\theta} = \{\sigma^2, M, \mathbf{a}, \mathbf{K}, \mathbf{s}, \mathbf{t}, \mathbf{v}\}$ where $\mathbf{K}$ is the M-vector of interaction degrees, $\mathbf{s}$ is the vector of signs $\{\{s_{km}\}_{k=1}^{K_m}\}_{m=1}^{M}$, $\mathbf{t}$ the vector of knots and $\mathbf{v}$ the vector of variables used (with $\mathbf{t}$ and $\mathbf{v}$ defined similar to $\mathbf{s}$). Under the Bayesian formulation, we specify a prior distribution for $\boldsymbol{\theta}$.

First, consider the priors for the σ^2 and $\mathbf{a}$. Let $\mathbf{B}$ be the $n \times (M+1)$ matrix of basis functions (including the intercept). Then we use Zellner's g-prior (Liang, Paulo, Molina, Clyde, and Berger 2008) for $\mathbf{a}$ with

$$\mathbf{a}|\sigma^2, \tau, \mathbf{B} \sim N(\mathbf{0}, \sigma^2(\mathbf{B}'\mathbf{B})^{-1}/\tau) \tag{4}$$

$$\sigma^2 \sim InvGamma(g_1, g_2) \tag{5}$$

$$\tau \sim Gamma(a_\tau, b_\tau) \tag{6}$$

with default settings $a_\tau = 1$ and $b_\tau = 1/n$ (shape and rate) to center the prior over the unit information prior and $g_1 = g_2 = 0$ resulting in the non-informative prior $p(\sigma^2) \propto 1/\sigma^2$. In practice, the default settings are sufficient for most cases, though it can be helpful to encode actual prior information into the prior for σ^2.

Now, consider the prior for the number of basis functions, M. We use a Poisson prior for M, truncated to be between 0 and $M_{\max}$. We give a Gamma hyperprior to the mean of the Poisson, λ. If c is the Poisson mass that has been truncated, i.e., $c = 1 - \sum_{m=0}^{M_{\max}} e^{-\lambda}\lambda^m/m!$, then we have

$$p(M|\lambda) = \frac{e^{-\lambda}\lambda^M}{cM!} \tag{7}$$

$$\lambda \sim Gamma(h_1, h_2) \tag{8}$$

where the default settings of $h_1 = h_2 = 10$ (shape and rate) in most cases induce a small number of basis functions. In practice, these hyperparameters can be key in order to prevent overfitting. More specifically, we increase h_2 (by many orders of magnitude in some cases) to bring the prior for λ very close to zero in an effort to thin out the tails of the Poisson and have fewer basis functions. We use $M_{\max}$ to give an upper bound to the computational cost, rather than to prevent overfitting. This strategy results in better fitting models since setting $M_{\max}$ too small often results in posterior sampling from only one mode.

The priors for $\mathbf{K}$, $\mathbf{s}$, $\mathbf{t}$ and $\mathbf{v}$ are uniform over a constrained space as described in Francom *et al.* (2016). The constraint in this prior makes sure basis functions have more than b non-zero values. Note that a basis function, as shown in Equation 3, is likely to have many zeros in it depending on how close the knot is to the edge of the space. If a knot is too close to the edge of the space, there might only be a few non-zero values in the basis function. A basis function with only a few non-zero values corresponds to very local fitting and usually results in edge effects. If we calculate the number of non-zero points in basis function m to be b_m, this prior requires that $b_m > b$. This is the BASS equivalent of specifying a minimum number of points in each partition in recursive partitioning. In addition to specifying b, we also specify $K_{\max}$, the maximum degree of interaction for each basis function.

Table 1 shows the parameters used in the `bass` function that we have discussed thus far, and what their mathematical symbols are.

symbol	$K_{\max}$	b	h_1	h_2	g_1	g_2	α	$M_{\max}$	a_τ	b_τ
`bass` function	`maxInt`	`npart`	`h1`	`h2`	`g1`	`g2`	`degree`	`maxBasis`	`a.tau`	`b.tau`

Table 1: Translation from mathematical symbols to parameters used in `bass` function.

2.1. Efficient Posterior Sampling

Posterior sampling is complicated by the fact that the model is transdimensional (since M is unknown). Our RJMCMC scheme allows us to add, delete, or change a basis function consistent with the approach of Nott *et al.* (2005). That is, instead of proposing to add a completely random new basis function in a reversible jump step, we use a proposal generating distribution that favors the variables and degrees of interaction already included in the model. For example, say there were five basis functions already in the model, each with degree of interaction two. Say the maximum degree of interaction was three. Then if we were proposing a new basis function we would sample the degree of interaction from $\{1, 2, 3\}$ with weights $\{w_1, w_1 + 5, w_1\}$, thus favoring two way interactions since we have seen more of them. If the nominal weight w_1 is large compared to the number of basis functions, this distribution looks more uniform. The value w_2 is the equivalent nominal weight for sampling variables to be included in a candidate basis function. Both w_1 and w_2 default to five. If there are a large number of unimportant variables in the data, a small value of w_2 (relative to M) helps to make posterior sampling more efficient by not proposing basis functions that include the unimportant variables.

We extend the framework of Nott *et al.* (2005) to allow for more than two-way interactions. This ends up being non-trivial, since the RJMCMC acceptance ratio requires us to calculate the probability of sampling the proposed basis function. The difficulty comes when we try to calculate the probability of sampling the particular variables, as this requires calculating the probability of a weighted sample without replacement (weighted since we do not sample variables uniformly, without replacement since variables cannot be used more than once in the same basis function). This is equivalent to sampling from the multivariate Wallenius' noncentral hypergeometric distribution. To determine the probability of such a sample, we use a function from the R package **BiasedUrn** (Fog 2015). Since the CRAN version of **BiasedUrn** allows for only 32 possible variables, we include a slightly altered version of the function in **BASS** to quickly evaluate the approximate density function of the multivariate Wallenius' noncentral hypergeometric distribution.

The computation behind posterior sampling becomes much more efficient when we recognize that each RJMCMC iteration only allows slight changes to our set of basis functions. Thus, quantities like $\mathbf{B}'\mathbf{B}$, $\mathbf{Ba}$ and $\mathbf{B}'\mathbf{y}$ can easily be updated rather than recalculated, as shown in Francom *et al.* (2016).

We perform N_{MCMC} RJMCMC iterations and discard the first N_{burn}, after which every N_{thin} iterations is kept. This results in $(N_{\text{MCMC}} - N_{\text{burn}})/N_{\text{thin}}$ posterior samples. Table 2 shows the parameters to the `bass` function discussed in this section, as well as their mathematical symbols.

symbol	w_1	w_2	N_{MCMC}	N_{burn}	N_{thin}
`bass` function	`w1`	`w2`	`nmcmc`	`nburn`	`thin`

Table 2: Translation from mathematical symbols to parameters used to specify nominal weights of proposal distributions and number of RJMCMC iterations in the `bass` function.

2.2. Parallel Tempering

Posterior sampling with RJMCMC is prone to mixing problems (problems exploring all of the parameter space). In our case, this is because only slight changes to the basis functions can be made in each iteration. Thus, once we start sampling from one mode of the posterior, it can be hard to move to another mode if it requires changing many of the basis functions (Gramacy, Samworth, and King 2010).

We are able to achieve better mixing by using parallel tempering. This requires the specification of a temperature ladder, $1 = t_1 < t_2 < \cdots < t_T < \infty$. For each temperature in the temperature ladder, a RJMCMC chain samples the posterior raised to the inverse temperature (i.e., if $\pi(\boldsymbol{\theta}|\mathbf{y})$ is the posterior of interest, we sample from $\pi(\boldsymbol{\theta}|\mathbf{y})^{1/t_i}$). The chains at neighboring temperatures are allowed to swap states according to a Metropolis-Hastings acceptance ratio (see Francom *et al.* (2016) and references therein). Only samples in the lowest temperature chain (t_1) are used for inference. The high temperature chains mix over many posterior modes, allowing diverse models to be propagated to the low temperature chain. We allow the chains to run without swapping for N_{st} iterations at the beginning of the run to allow them to get close to their stationary distributions.

Specifying a temperature ladder can be difficult. Temperatures need to be close enough to each other to allow for frequent swaps (with acceptance rates between 20 and 60% (Altekar, Dwarkadas, Huelsenbeck, and Ronquist 2004)), and the highest temperature (t_T) needs to be high enough to be able to explore all the modes. Future versions of this package may make some attempt at automatically specifying and altering a temperature ladder. Further, a message passing interface (MPI) approach to handling the multiple chains could result in substantial speedup, and may be implemented in future versions of the package.

Table 3 shows the translation from parameters used for parallel tempering in the `bass` function to symbols we have used in this section.

symbol	$(t_1, \ldots, t_T)$	N_{st}
`bass` function	`temp.ladder`	`start.temper`

Table 3: Translation from mathematical symbols to parameters used for parallel tempering in the `bass` function.

2.3. Functional Response

We handle Functional responses as though the variable indexing the functional response, like time or location, is one of the independent variables. When the functional response is output onto the same functional variable grid for all samples, this results in more efficient

calculations involving basis functions because of the Khatri-Rao product structure (Francom *et al.* 2016). For example, this software is well suited to fit a model where the data are such that a combination of independent variables results in a time-series and the grid of times (say, $r_1, \ldots, r_q$) is the same for each combination.

If there are multiple functional variables, we must specify a maximum degree of interaction for them, $K^F_{\max}$. For instance, if the functional output was a spatiotemporal field (a function of three variables) and we specify a maximum degree of functional interaction of two, we would not allow for interactions between both spatial dimensions and time. We would specify the grid of spatial locations and time points as a matrix with three columns rather than a vector like we did in the time series example above. We can also specify a value b_F, possibly different from b, that indicates the number of non-zero values required in the functional part of basis functions. When functional responses are included, the values of b and b_F should be relative to the sample size and the size of the functional grid, respectively.

Table 4 shows parameters necessary to model functional responses in the `bass` function. The response `y` should be specified as a matrix when the response is functional.

symbol	$(r_1, \ldots, r_q)$	$K^F_{\max}$	b_F
`bass` function	`xx.func`	`maxInt.func`	`npart.func`

Table 4: Translation from mathematical symbols to parameters used in the `bass` function when modeling functional data.

2.4. Categorical Inputs

We include categorical variables by allowing for basis functions to include indicators for categorical variables being in certain categories. Our approach is the Bayesian version of Friedman (1991a) and is described in Francom, Sansó, Bulaevskaya, and Lucas (2017). If a set of independent variables is separated into continuous variables $\mathbf{x}$ and categorical variables $\mathbf{c}$, then the m^{th} basis function equivalent of Equation 3 can be written as

$$B_m(\mathbf{x}, \mathbf{c}) = \prod_{k=1}^{K_m} g_{km}[s_{km}(x_{v_{km}} - t_{km})]_+^{\alpha} \prod_{l=1}^{K^c_m} 1\left(c_{v^c_{lm}} \in C_{lm}\right) \tag{9}$$

where K^c_m is the degree of interaction for the categorical predictors, $1(\cdot)$ is the indicator function, v^c_{lm} indexes the categorical variables and C_{lm} is a subset of the categories for variables $c_{v^c_{lm}}$. We now allow for K_m or K^c_m to be zero, and specify a $K^c_{\max}$ (`maxInt.cat` in the `bass` function).

The priors we use for the degree of interaction, variables used and categories used are, in combination with the priors we used above, the same constrained uniform. Thus, basis function $(B_m(\mathbf{x}_1, \mathbf{c}_1), \ldots, B_m(\mathbf{x}_n, \mathbf{c}_n))$ is required to have at least b non-zero values.

3. Sensitivity Analysis

Global sensitivity analysis for nonlinear models using the Sobol' decomposition (Sobol' 2001) is well developed, but often requires large numbers of evaluations of the models for Monte

Carlo approximation of integrals (Saltelli, Ratto, Andres, Campolongo, Cariboni, Gatelli, Saisana, and Tarantola 2008). The benefit of polynomial spline models is that Monte Carlo approximation is unnecessary because the integrals can be calculated analytically.

The method decomposes a function $f(\mathbf{x})$ into main effects, two way interactions, and so on, up to p way interactions so that

$$f(\mathbf{x}) = f_0 + \sum_{i=1}^{p} f_i(x_i) + \sum_{i=1}^{p}\sum_{j>i} f_{ij}(x_i, x_j) + \cdots + f_{1\cdots p}(x_1, \ldots, x_p). \tag{10}$$

Each term in the sum above is constructed so that it is orthogonal to all the other terms. This can be done by calculating

$$f_0 = \int f(\mathbf{x}) d\mathbf{x} \tag{11}$$

$$f_i(x_i) = \int f(\mathbf{x}) d\mathbf{x}_{-i} - f_0 \tag{12}$$

$$f_{ij}(x_i, x_j) = \int f(\mathbf{x}) d\mathbf{x}_{-ij} - f_i(x_i) - f_j(x_j) - f_0 \tag{13}$$

etc., for all the terms in Equation 10. Note that if we assume $\mathbf{x}$ is uniformly distributed, these are conditional expectations (except f_0 is unconditional). The conditional expectations are centered at zero the way we have constructed them. Since the terms in Equation 10 are orthogonal,

$$E(f^2(\mathbf{x})) = f_0^2 + \sum_{i=1}^{p} E\left(f_i^2(x_i)\right) + \sum_{i=1}^{p}\sum_{j>i} E\left(f_{ij}^2(x_i, x_j)\right) + \cdots + E\left(f_{1\cdots p}^2(x_1, \ldots, x_p)\right). \tag{14}$$

Using the fact that $Var(f(\mathbf{x})) = E(f^2(\mathbf{x})) - f_0^2$ and that $E(f_{i_1\cdots i_s}(x_{i_1\cdots i_s})) = 0$ for all terms except f_0,

$$Var(f(\mathbf{x})) = \sum_{i=1}^{p} Var(f_i(x_i)) + \sum_{i=1}^{p}\sum_{j>i} Var(f_{ij}(x_i, x_j)) + \cdots + Var(f_{1\cdots p}(x_1, \ldots, x_p)). \tag{15}$$

This is a decomposition of the variance of the model into variance due to each main effect, each two way interaction (after accounting for the associated main effects), etc. All of these integrals are analytical in our case, with solutions given in Francom *et al.* (2016). Sensitivity indices for main effects and interactions are then defined as proportions of the total variance. Total sensitivity for a particular variable can then be gauged by adding the main effect and all interactions associated with that variable and comparing to the total sensitivity indices for other variables.

We can obtain this variance decomposition for each posterior sample to get posterior distributions of sensitivity indices. This can be time consuming, so the `sobol` function has an argument `mcmc.use` to specify which RJMCMC iterations should be used. Calculations of the integrals above can be vectorized when basis functions are the same and only basis function coefficients change. This is the case for many of the RJMCMC iterations, and the `sobol` function automatically determines this and accounts for it. (As a side note, this is also the case for the `predict` function).

3.1. Functional Response

There are a few ways to think about sensitivity analysis for models with functional response. One way is to get the sensitivity indices for the functional variables in the same way we get the sensitivity indices for the rest of the variables. This results in a total variance decomposition. Another approach is to obtain functional sensitivity indices, which would tell us how important a variable or interaction is as we change the functional variable. This can be done by following the procedure just mentioned, but simply not integrating over the functional variable. Hence, all of the expectations above would be conditional on the functional variable. These approaches are explored in Francom *et al.* (2016).

By default, the `sobol` function gets sensitivity indices for the functional variables the same way it does for the other variables. Setting `func.var = 1` gets the sensitivity indices as functions of the first (possibly only) functional variable (if there are multiple functional variables, this refers to the first column of the matrix `xx.func` passed to the `bass` function).

3.2. Categorical Inputs

Under our categorical input extension, the necessary expectations to obtain the Sobol' decomposition are still analytical, as described in Francom *et al.* (2017). For the categorical variables, we replace the integrals with sums over categories.

4. Examples

We now demonstrate the capabilities of the package on a few examples. For each example, we start by setting the seed (`set.seed(0)`) so that readers can replicate the results. First we load the package

```
R> library("BASS")
```

which we use for all the examples.

4.1. Curve Fitting

We first demonstrate how the package can be used for curve fitting. We generate $y \sim N(f(x), 1)$ where $x \in [-5, 5]$ and

$$f(x) = \begin{cases} -0.1x^3 + \sin(\pi x^2)(x-4)^2 & 0 < x < 4 \\ -0.1x^3 & \text{otherwise} \end{cases} \tag{16}$$

for 1000 samples of x. The data are shown in Figure 3.

We generate the data with the following code.

```
R> set.seed(0)
R> f <- function(x) {
+   -.1 * x^3 + 2 * as.numeric((x < 4) * (x > 0)) * sin(pi * x^2) * (x - 4)^2
+ }
R> sigma <- 1
```

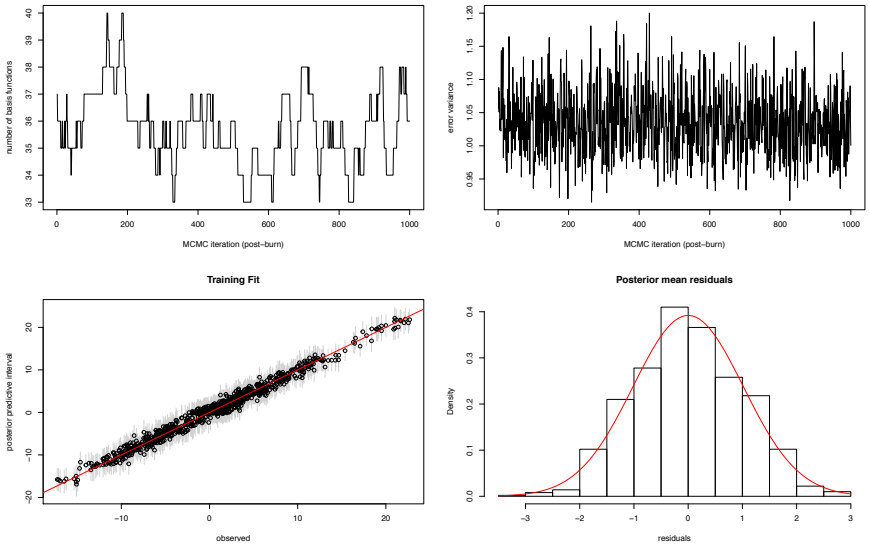

Figure 1: Diagnostic plots for BASS model fitting.

```
R> n <- 1000
R> x <- runif(n, -5, 5)
R> y <- rnorm(n, f(x), sigma)
```

We then call the `bass` function to fit a BASS model using the default settings.

```
R> mod<-bass(x, y)

MCMC Start #--Mon Jan  9 16:58:48 2017--# nbasis: 0
MCMC iteration 1000 #--Mon Jan  9 16:58:50 2017--# nbasis: 33
MCMC iteration 2000 #--Mon Jan  9 16:58:52 2017--# nbasis: 33
MCMC iteration 3000 #--Mon Jan  9 16:58:54 2017--# nbasis: 32
MCMC iteration 4000 #--Mon Jan  9 16:58:57 2017--# nbasis: 34
MCMC iteration 5000 #--Mon Jan  9 16:58:59 2017--# nbasis: 31
MCMC iteration 6000 #--Mon Jan  9 16:59:01 2017--# nbasis: 39
MCMC iteration 7000 #--Mon Jan  9 16:59:03 2017--# nbasis: 31
MCMC iteration 8000 #--Mon Jan  9 16:59:05 2017--# nbasis: 33
MCMC iteration 9000 #--Mon Jan  9 16:59:07 2017--# nbasis: 38
MCMC iteration 10000 #--Mon Jan  9 16:59:09 2017--# nbasis: 36
```

The result is an object that can be used for prediction and sensitivity analysis. By default, the bass function prints progress after each 1000 MCMC iterations, along with the number of basis functions. To diagnose the fit of the model, we call the plot function.

```
R> plot(mod)
```

This generates the four plots shown in Figure 1. The top left and right plots show trace plots (after burn-in and excluding thinned samples) of the number of basis functions (M) and the

error variance (σ^2). The bottom left plot shows the response values plotted against the the posterior mean predictions (with equal tail posterior probability intervals as specified by the `quants` parameter). The bottom right plot shows a histogram of the posterior mean residuals along with the assumed Gaussian distribution centered at zero and with variance taken to be the posterior mean of σ^2. This is for checking the Normality assumption.

Next, we can generate posterior predictions at new inputs, which we generate as `x.test`.

```
R> n.test <- 1000
R> x.test <- sort(runif(n.test, -5, 5))
R> pred <- predict(mod, x.test, verbose = T)

Predict Start #--Mon Jan  9 16:59:10 2017--# Models: 164
Predict #--Mon Jan  9 16:59:10 2017--# Model: 100
```

By default, the `predict` function generates posterior predictive distributions for all of the inputs. We can use a subset of posterior samples by specifying the parameter `mcmc.use`. For instance, `mcmc.use = 1` will use the first posterior sample (after burn-in and excluding thinned samples), and will thus be faster. Rather than iterating through the MCMC samples to generate predictions, we instead iterate through "models." The model changes when the basis functions change, which means that we can build the basis functions once and perform vectorized operations for predictions for all the MCMC iterations with the same basis functions.

The object resulting from the `predict` function is a matrix with rows corresponding to MCMC samples and columns corresponding to settings of `x.test`. Thus, the posterior mean predictions are obtained by taking the column means. We plot the posterior predictive means against the true values of $f(x)$ as shown in Figure 2.

```
R> fx.test <- f(x.test)
R> plot(fx.test, colMeans(pred))
R> abline(a = 0, b = 1, col = 2)
```

Note that the predictive distributions in the columns of `pred` are for $f(x)$. To obtain predictive distributions for data, we would need to include Gaussian error with variance σ^2 (demonstrated in Section 4.5). Posterior samples of σ^2 are given in `mod$s2`.

In the curve fitting case, we can plot predicted curves. Below, we plot 10 posterior predictive samples along with the true curve (Figure 3). We also show knot locations (in the rug along the x-axis) for one of the posterior samples.

```
R> plot(x, y, cex = .5)
R> curve(f(x), add = T, lwd = 3, n = 1000, col = 2, lty = 2)
R> matplot(x.test, t(pred[seq(100, 1000, 100), ]), type='l', add=T, col=3)
R> rug(BASS:::unscale.range(mod$curr.list[[1]]$knots.des, range(x)))
R> legend('topright', legend = c('true curve', 'posterior predictive draws'),
+        col = c(2:3), lty = c(2, 1), lwd = c(3, 1), bty = 'n')
```

If we are interested in using fewer knots (fewer basis functions), we can change the prior for the number of basis functions to be more restrictive. For instance, setting `h2=100`

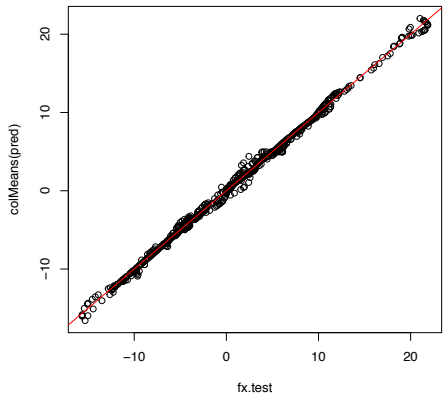

Figure 2: BASS prediction on test data.

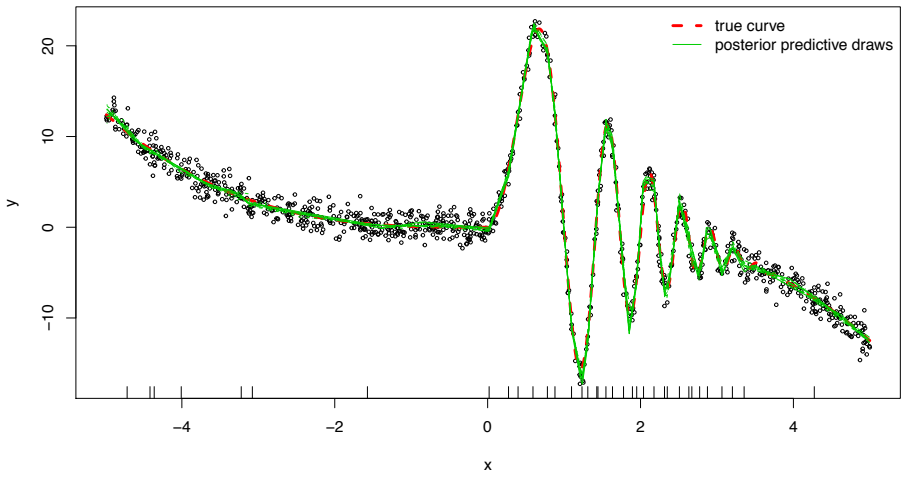

Figure 3: True curve with posterior predictive draws.

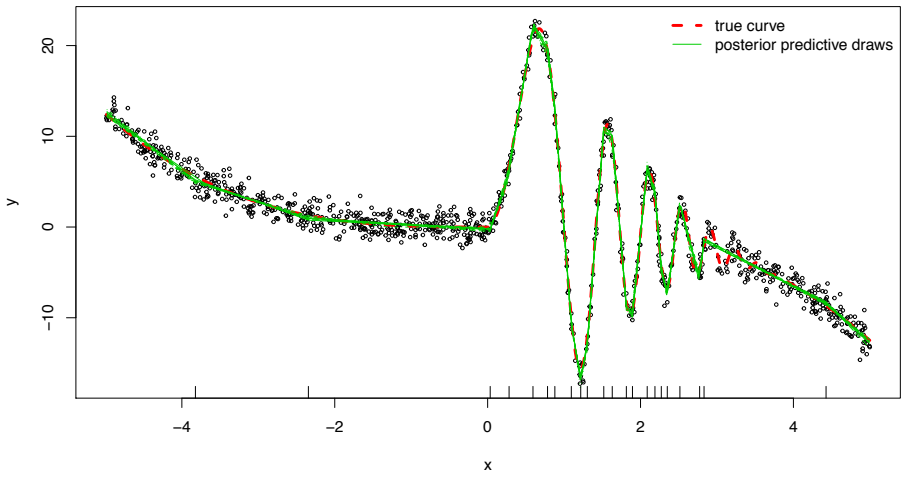

Figure 4: True curve with posterior predictive draws and more restrictive prior on the number of basis functions.

```
R> mod <- bass(x, y, h2 = 100)
```

```
R> pred <- predict(mod, x.test)
R> plot(x, y, cex = .5)
R> curve(f(x), add = T, lwd = 3, n = 1000, col = 2, lty = 2)
R> matplot(x.test, t(pred[seq(100, 1000, 100), ]), type='l', add=T, col=3)
R> rug(BASS:::unscale.range(mod$curr.list[[1]]$knots.des, range(x)))
R> legend('topright', legend = c('true curve', 'posterior predictive draws'),
+        col = c(2:3), lty = c(2, 1), lwd = c(3, 1), bty = 'n')
```

results in knots as shown along the x-axis of Figure 4. This results in fewer knots, but perhaps slight underfitting in the part of the curve around $x = 3$. The `h2` parameter can be used to prevent overfitting, but the setting is not intuitive. Thus, this parameter may require tuning (perhaps by cross-validation).

Two final issues to discuss with this example are why we use linear splines (the default `degree = 1`) and how to tell if we have achieved convergence before taking MCMC samples as posterior samples. We use linear splines almost exclusively when using this package because of their stability and ability to capture nonlinear curves and surfaces. Using a higher degree, such as `degree = 3`, results in smoother models but suffers from stability problems and is more difficult to fit. We suggest settings of `degree` other than `degree = 1` be used with care, always with scrutiny of prediction performance. Convergence is best assessed by examining the trace plots shown in Figure 1. Especially if the trace plot for σ^2 shows any sort of non-cyclical pattern, the sampler should be run for longer. As a side note, a new sampler can be started from where the old sampler left off by using the `curr.list` parameter. For instance, we can run `mod2 <- bass(x, y, curr.list = mod$curr.list)` to start a new sampler from where `mod` left off.

4.2. Friedman Function

For our next example, we will test the package on the Friedman function (Friedman 1991b). This function will have 10 inputs, five of which contribute nothing. The other five are used to generate

$$f(\mathbf{x}) = 10\sin(\pi x_1 x_2) + 20(x_3 - 0.5)^2 + 10x_4 + 5x_5. \tag{17}$$

We generate 200 input samples uniformly from a unit hypercube, calculate $f(x)$ for each and add standard Normal error to obtain data to model.

```
R> set.seed(0)
R> f <- function(x) {
+   10 * sin(pi * x[, 1] * x[, 2]) + 20 * (x[, 3] - .5)^2 +
+     10 * x[, 4] + 5 * x[, 5]
+ }
R> sigma <- 1
R> n.vars <- 10
R> n <- 200
R> x <- matrix(runif(n * n.vars), n, n.vars)
R> y <- rnorm(n, f(x), sigma)
```

Here we will show how we can change the length of the MCMC chain and use parallel tempering. We run the RJMCMC chain for 40000 iterations, discarding the first 30000 as burn-in and thinning by keeping every tenth sample. We supply a temperature ladder with smallest value one (the "cold chain", or true posterior) and largest value 8.15 (the "hottest" chain) using geometric spacing. Thus, $t_i = (1 + \Delta_t)^{i-1}$ where Δ_t is a spacing parameter we set at 0.3. We use nine chains. By default, chains at neighboring temperatures will be allowed to swap after the first 1000 iterations.

```
R> mod <- bass(x, y, nmcmc = 40000, nburn = 30000, thin = 10,
+              temp.ladder = (1 + .3)^(1:9 - 1), verbose = F)
```

We can generate posterior predictive samples just as we did in the curve fitting example.

```
R> n.test <- 1000
R> x.test <- matrix(runif(n.test * n.vars), n.test)
R> pred <- predict(mod, x.test, verbose = T)

Predict Start #--Mon Jan  9 19:24:31 2017--# Models: 828
Predict #--Mon Jan  9 19:24:31 2017--# Model: 100
Predict #--Mon Jan  9 19:24:32 2017--# Model: 200
Predict #--Mon Jan  9 19:24:32 2017--# Model: 300
Predict #--Mon Jan  9 19:24:32 2017--# Model: 400
Predict #--Mon Jan  9 19:24:32 2017--# Model: 500
Predict #--Mon Jan  9 19:24:32 2017--# Model: 600
Predict #--Mon Jan  9 19:24:32 2017--# Model: 700
Predict #--Mon Jan  9 19:24:32 2017--# Model: 800
```

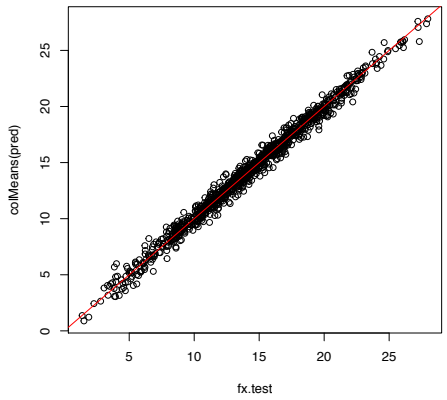

Figure 5: BASS prediction on test data - Friedman function.

Plotting these samples against true values of $f(x)$ shows that we have a good fit (Figure 5).

```
R> fx.test <- f(x.test)
R> plot(fx.test, colMeans(pred))
R> abline(a = 0, b = 1, col = 2)
```

Now that we are considering a function of many variables, we may be interested in sensitivity analysis. To get the Sobol' decompostion for each posterior sample, we use the `sobol` function.

```
R> sens <- sobol(mod, verbose = F)
```

Note that when `verbose = T`, this function prints after every 10 models (as with the `predict` function, vectorizing around models rather than MCMC iterations saves a large amount of time). Depending on the number of basis functions and the number of models, this function can take significant amounts of time. If that is the case, using a smaller set of MCMC iterations by specifying `mcmc.use` may be useful.

The default plotting for this kind of object (Figure 6) shows boxplots of variance explained for each main effect and interaction that shows up in the BASS model. It also shows boxplots of the total sensitivity indices.

```
R> plot(sens, cex.axis = .5)
```

If there are a large number of main effects or interactions that explain very small percentages of variation, we can show only the effects that are most significant. For instance, we could show only the effects that, on average, explain at least 1% of the variance (Figure 7).

```
R> boxplot(sens$S[, colMeans(sens$S) > .01], las = 2,
+         ylab = 'proportion variance', range = 0)
```

As expected, we see that almost all of the variance is from the first five variables and the only strong interaction is between the first two variables.

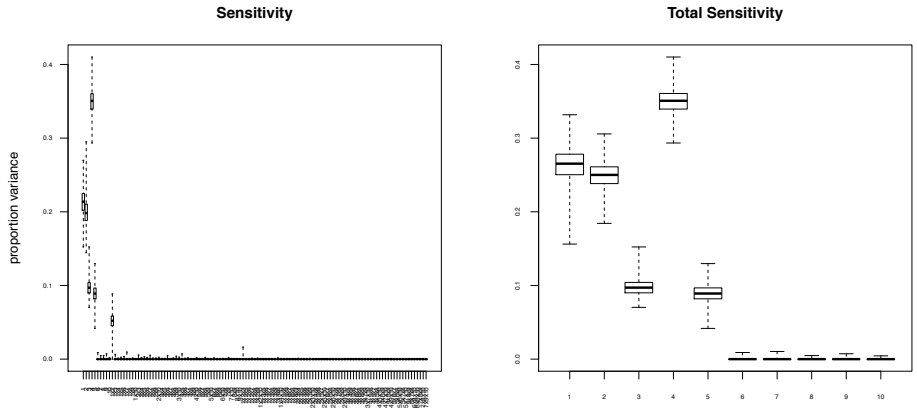

Figure 6: BASS sensitivity analysis - Friedman function.

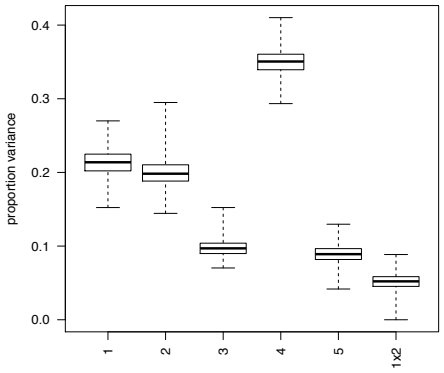

Figure 7: Most important main effects and interactions - Friedman function.

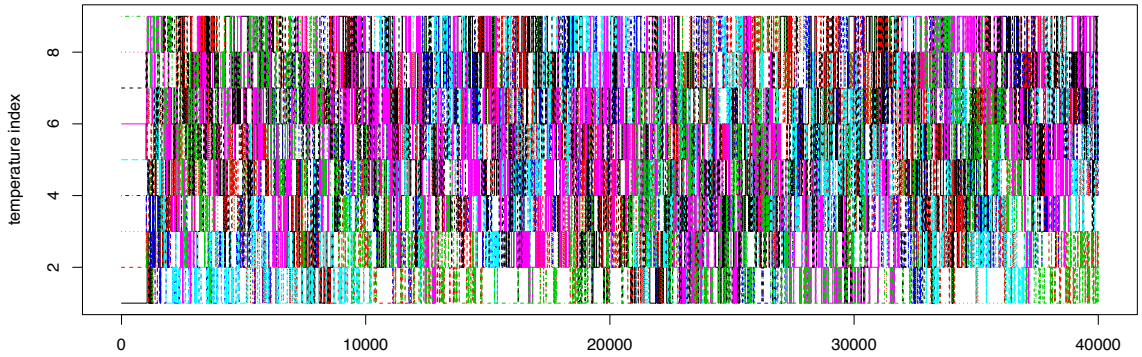

Figure 8: Parallel tempering diagnostics - swap trace plot.

As a final note for this example, we discuss tempering diagnostics. We would like for neighboring chains to have swap acceptance rate of somewhere around 23%. Running `bass` with `verbose = T` prints these acceptance rates every 1000 iterations. At the completion of the sampling, we can investigate acceptance rates by dividing the swap counts by the number of swap proposals, as follows.

```
R> mod$count.swap/mod$count.swap.prop

[1] 0.1786158 0.2653317 0.2911800 0.3111511 0.3255474 0.3274227 0.3084760
[8] 0.2305950
```

Since we have specified nine temperatures, there are eight possible swaps, hence the eight numbers. If, for example, we wanted to increase the first acceptance rate, we would move the second temperature closer to the first.

Further analysis of swaping can be done by looking at swap trace plots.

```
R> matplot(mod$temp.val, type = 'l', ylab = 'temperature index')
```

Figure 8 shows the swap trace plot where y-axis values are temperature indices (1 is the true posterior and 9 is the posterior raised to the smallest power), the x-axis shows MCMC iteration and the colored lines represent the different chains. We want to see these chains mixing throughout, as we do here.

Determining whether the smallest value of the temperature ladder is small enough to allow for good mixing can be difficult. In this example, we could run the model with `temp.ladder = 8.15` and look at mixing diagnostics. One could also look at predicted versus observed plots at the different temperatures for the last MCMC iteration by executing the following code, the output of which is shown in Figure .

```
R> par(mfrow=c(3,3))
R> temp.ind <- sapply(mod$curr.list, function(x) x$temp.ind)
```

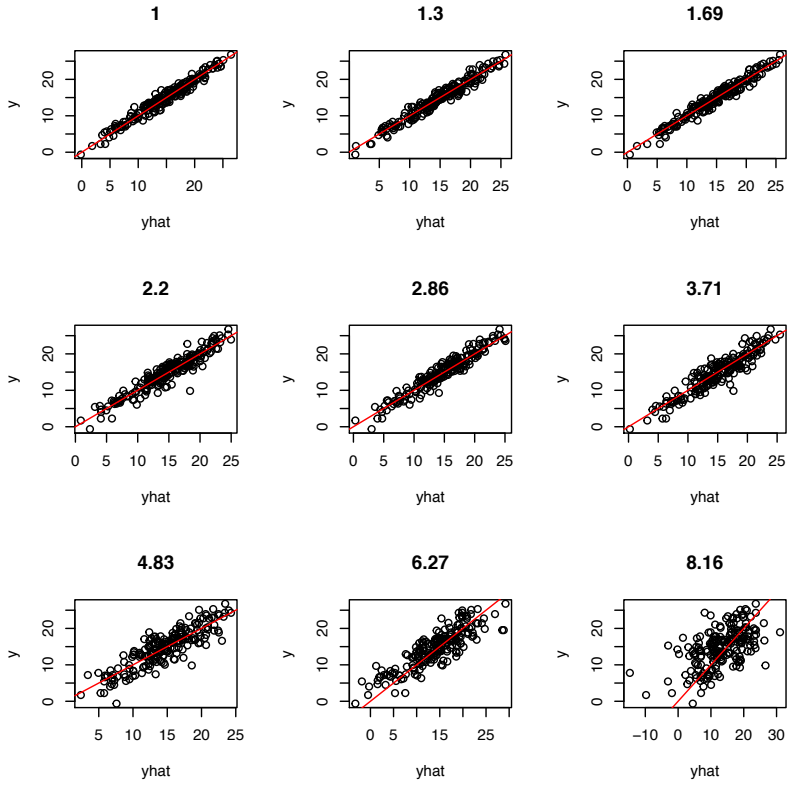

Figure 9: Predicted versus observed for the last MCMC iteration of the nine chains at different temperatures. The temperatures are shown above each plot.

```
R> for(i in 1:length(mod$temp.ladder)) {
+   ind <- which(temp.ind == i)
+   yhat <- mod$curr.list[[ind]]$des.basis %*% mod$curr.list[[ind]]$beta
+   plot(yhat, y, main = round(mod$temp.ladder[i], 2))
+   abline(a = 0, b = 1, col = 2)
+ }
```

Note that the `curr.list` object is a list with number of elements equal to the number of temperatures. This list contains the MCMC state for each chain. Since we swap temperatures rather than entire states, the chains are not in order according to temperature. We note that using the default prior for σ^2 with a temperature ladder with relatively large values can lead to instabilities when estimating σ^2. In cases where that is clearly the case, the prior for σ^2 will be automatically changed and a warning will be generated.

To demonstrate what is different when we use tempering, consider the equivalent BASS model fit without tempering.

```
R> mod.noTemp <- bass(x, y, nmcmc = 40000, nburn = 30000,
+                     thin = 10, verbose = F)
```

We compare the root mean square prediction error (RMSE) for the two models, as well as the

empirical coverage of 95% probability intervals. First, the RMSE for the model fit without tempering

```
R> pred.noTemp <- predict(mod.noTemp, x.test)
R> sqrt(mean((colMeans(pred.noTemp) - fx.test)^2))

[1] 0.6366317
```

and the empirical coverage

```
R> quants.noTemp <- apply(pred.noTemp, 2, quantile, probs = c(.025, .975))
R> mean((quants.noTemp[1, ] < fx.test) & (quants.noTemp[2, ] > fx.test))

[1] 0.841
```

demonstrate that the fit is quite good. When we use parallel tempering, the RMSE

```
R> sqrt(mean((colMeans(pred) - fx.test)^2))

[1] 0.5334709
```

and the empirical coverage

```
R> quants <- apply(pred, 2, quantile, probs = c(.025, .975))
R> mean((quants[1, ] < fx.test) & (quants[2, ] > fx.test))

[1] 0.972
```

are better, though not by an extreme amount. Under different seeds, we tend to see higher coverage when we use tempering and lower coverage when we do not. We also tend to get better models in terms of RMSE when we use tempering. Other benefits of tempering will be shown in later examples. Because the computational burden is currently linear in the number of temperatures, using fewer temperatures is better. Thus, for many purposes, the model without tempering may be good enough.

4.3. Friedman Function with a Categorical Variable

In this example, we use data generated from a function similar to the Friedman function in the previous example but with a categorical variable included. The function, introduced in Gramacy and Taddy (2010), has

$$f(\mathbf{x}) = \begin{cases} 10\sin(\pi x_1 x_2) & x_{11} = 1 \\ 20(x_3 - 0.5)^2 & x_{11} = 2 \\ 10x_4 + 5x_5 & x_{11} = 3 \\ 5x_1 + 10x_2 + 20(x_3 - 0.5)^2 + 10\sin(\pi x_4 x_5) & x_{11} = 4 \end{cases} \tag{18}$$

as the mean function and standard Normal error. Again, $x_6, \ldots, x_{10}$ are unimportant. We generate 500 random uniform samples of the first 10 variables and randomly sample 500 values of the four categories of the 11^{th} variable. The `bass` function treats input variables as categorical only if they are coded as factors.

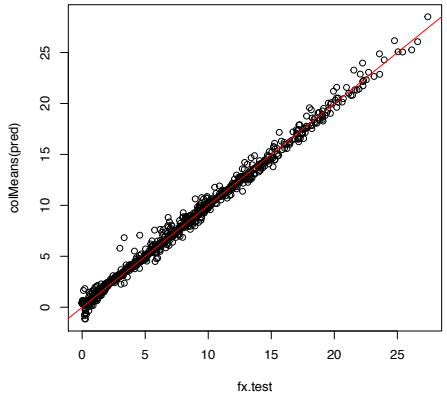

Figure 10: BASS prediction on test data - Friedman function with categorical predictor.

```
R> set.seed(0)
R> f <- function(x) {
+   as.numeric(x[, 11] == 1) * (10 * sin(pi * x[, 1] * x[, 2])) +
+   as.numeric(x[ ,11] == 2) * (20 * (x[, 3] - .5)^2) +
+   as.numeric(x[, 11] == 3) * (10 * x[, 4] + 5 * x[, 5]) +
+   as.numeric(x[, 11] == 4) * (10 * sin(pi * x[, 5] * x[, 4]) +
+                     20 * (x[, 3] - .5)^2 + 10 * x[, 2] + 5 * x[, 1])
+ }
R> sigma <- 1
R> n <- 500
R> x <- data.frame(matrix(runif(n * 10), n, 10),
+                   as.factor(sample(1:4, size = n, replace = T)))
R> y <- rnorm(n, f(x), sigma)
```

We fit a model with tempering and use it for prediction, as in the previous example.

```
R> mod <- bass(x, y, nmcmc = 40000, nburn = 30000, thin = 10,
+              temp.ladder = (1 + .2)^(1:6 - 1), verbose = F)
R> n.test <- 1000
R> x.test <- data.frame(matrix(runif(n.test * 10), n.test, 10),
+                       as.factor(sample(1:4, size = n.test, replace = T)))
R> pred <- predict(mod, x.test)
```

Plotting posterior predictive samples against true values of $f(x)$ shows that we have a good fit (Figure 10).

```
R> fx.test <- f(x.test)
R> plot(fx.test, colMeans(pred))
R> abline(a = 0, b = 1, col = 2)
```

Sensitivity analysis is performed in the same manner.

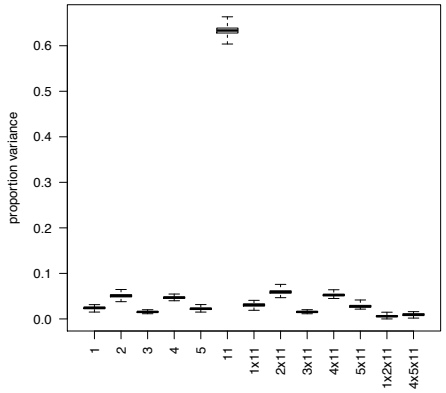

Figure 11: Most important main effects and interactions - Friedman function with categorical predictor.

```
R> sens <- sobol(mod)
```

Plotting the posterior distributions of the most important (explaining more than 0.5% of the variance) sensitivity indices in Figure 11, we see how important the categorical variable is as well as which variables it interacts with.

```
R> boxplot(sens$S[, colMeans(sens$S) > .005], las = 2,
+          ylab = 'proportion variance', range = 0)
```

4.4. Friedman Function with Functional Response

Next, we consider an extension of the Friedman function that is functional in one variable Francom *et al.* (2016). We use

$$f(\mathbf{x}) = 10\sin(2\pi x_1 x_2) + 20(x_3 - 0.5)^2 + 10x_4 + 5x_5 \tag{19}$$

where we treat x_1 as the functional variable. Note that we insert a two into the sin function in order to increase the variability due to x_1, making the problem more challenging. We generate 500 combinations of $x_2, \ldots, x_{10}$ from a uniform hypercube. We generate a grid of values of x_1 of length 50. This ends up being 500×50 combinations of inputs, for which we evaluate f and add standard Normal error. We keep the responses in a matrix of dimension 500×50 so that each row represents a curve. The inputs are kept separate in a 500×9 matrix and a grid of length 50.

```
R> set.seed(0)
R> f<-function(x) {
+   10 * sin(2 * pi * x[, 1] * x[, 2]) + 20 * (x[, 3] - .5)^2 +
+     10 * x[, 4] + 5 * x[, 5]
+ }
R> sigma <- 1
```

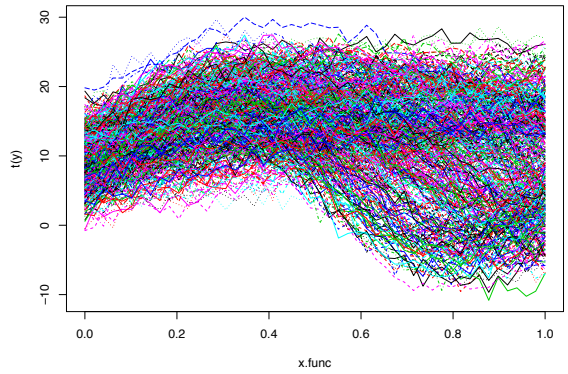

Figure 12: 500 Functional responses. The goal is to fit a functional nonparametric regression model and perform sensitivity analysis.

```
R> n <- 500
R> n.func <- 50
R> x.func <- seq(0, 1, length.out = n.func)
R> x <- matrix(runif(n * 9), n)
R> y <- matrix(f(cbind(rep(x.func, each = n),
+                      kronecker(rep(1, n.func), x))),
+              ncol = n.func) + rnorm(n * n.func, 0, sigma)
```

The functional data can be plotted as follows and are shown in Figure 12.

```
R> matplot(x.func, t(y), type='l')
```

In order for the **BASS** package to handle functional responses, each curve needs to be evaluated on the same grid. Thus, the responses must be able to be stored as a matrix without missing values.

We fit the model by specifying our matrices `x` and `y` as well as the grid `x.func`.

```
R> mod <- bass(x, y, xx.func = x.func)
```

```
MCMC Start #--Mon Jan  9 21:11:00 2017--# nbasis: 0
MCMC iteration 1000 #--Mon Jan  9 21:11:02 2017--# nbasis: 110
MCMC iteration 2000 #--Mon Jan  9 21:11:06 2017--# nbasis: 176
MCMC iteration 3000 #--Mon Jan  9 21:11:12 2017--# nbasis: 173
MCMC iteration 4000 #--Mon Jan  9 21:11:16 2017--# nbasis: 115
MCMC iteration 5000 #--Mon Jan  9 21:11:18 2017--# nbasis: 74
MCMC iteration 6000 #--Mon Jan  9 21:11:19 2017--# nbasis: 60
MCMC iteration 7000 #--Mon Jan  9 21:11:21 2017--# nbasis: 60
MCMC iteration 8000 #--Mon Jan  9 21:11:22 2017--# nbasis: 62
MCMC iteration 9000 #--Mon Jan  9 21:11:24 2017--# nbasis: 71
MCMC iteration 10000 #--Mon Jan  9 21:11:31 2017--# nbasis: 62
```

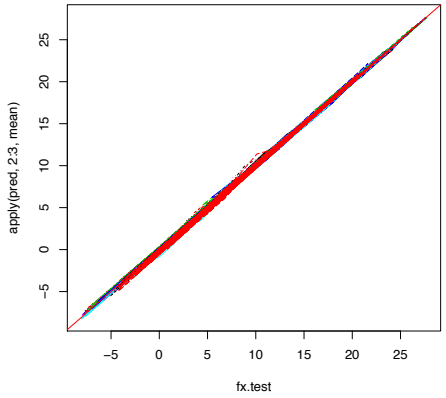

Figure 13: BASS prediction performance - Friedman function with functional response.

Prediction is as simple as before. If we want to predict on a different functional grid, we can specify that in the `predict` function with `newdata.func`.

```
R> n.test <- 100
R> x.test <- matrix(runif(n.test * 9), n.test)
R> pred <- predict(mod, x.test)
```

Following, we make a functional predicted versus observed plot, shown in Figure 13

```
R> fx.test<-matrix(f(cbind(rep(x.func, each = n.test),
+                          kronecker(rep(1, n.func), x.test))), ncol=n.func)
R> matplot(fx.test, apply(pred, 2:3, mean), type = 'l')
R> abline(a = 0, b = 1, col = 2)
```

We will demonstrate the two methods of sensitivity analysis discussed in Section 3. First, we can get the Sobol' indices for the functional variable and its interactions just as we do the other variables. This is the default.

```
R> sens <- sobol(mod, mcmc.use = 1:100)
```

```
Sobol Start #--Mon Jan  9 21:11:43 2017--# Models: 25
Sobol #--Mon Jan  9 21:11:57 2017--# Model: 10
Sobol #--Mon Jan  9 21:12:11 2017--# Model: 20
Total Sensitivity #--Mon Jan  9 21:12:18 2017--#
```

When we plot the variance decomposition, as shown in Figure 14, the functional variable is labeled with the letter "a." If we had multiple functional variables, they would be labeled with different letters.

```
R> plot(sens, cex.axis = .5)
```

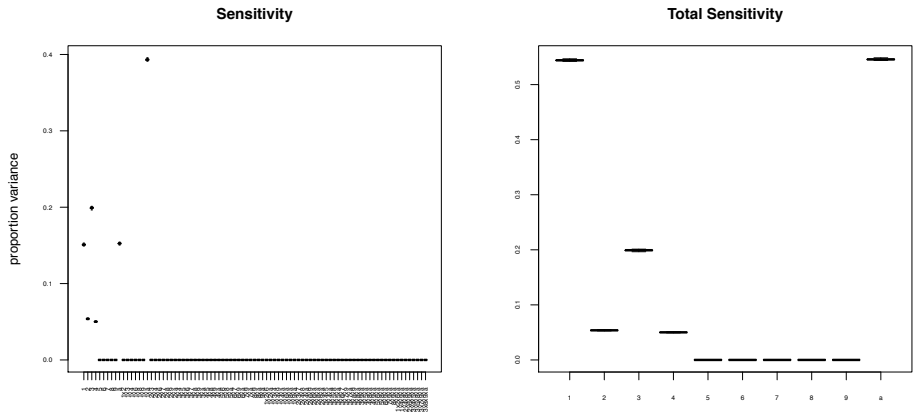

Figure 14: Sensitivity analysis - Friedman function with functional response.

The other approach to sensitivity analysis is to get a functional variance decomposition. This is done by using the `func.var` parameter. If there is only one functional variable, we set `func.var = 1`. Otherwise we set `func.var` to the column of `xx.func` we want to use for our functional variance decomposition. This will be explained in more detail in a later example.

```
R> sens.func <- sobol(mod, mcmc.use = 1:100, func.var = 1)

Sobol Start #--Mon Jan  9 21:12:19 2017--# Models: 25
Sobol #--Mon Jan  9 21:12:29 2017--# Model: 10
Sobol #--Mon Jan  9 21:12:39 2017--# Model: 20
```

When we plot the variance decomposition, shown in Figure 15, we we get two plots.

```
R> plot(sens.func, cex = .5)
```

The left plot shows the posterior mean (using posterior samples specified with `mcmc.use`) of the functional sensitivity indices in a functional pie chart. The right plot shows the variance decomposition as a function of the functional variable. Thus, the top line in the right plot is the total variance in y as a function of x_1. The bottom line (black) is the total variance explained by the main effect of x_2 as a function of x_1. The labels in the plot on the left are the variable numbers (columns of `x`).

4.5. Air Foil Data

In this example, we consider a NASA data set, obtained from a series of aerodynamic and acoustic tests of two and three-dimensional airfoil blade sections conducted in an anechoic wind tunnel (Lichman 2013). The response is scaled sound pressure level, in decibels. There are five inputs: (1) Frequency, in Hertzs; (2) angle of attack, in degrees; (3) chord length, in meters; (4) free-stream velocity, in meters per second and (5) suction side displacement thickness, in meters. The data have 1503 combinations of these inputs, some of which are collinear (variables 2 and 5 have correlation of 0.75).

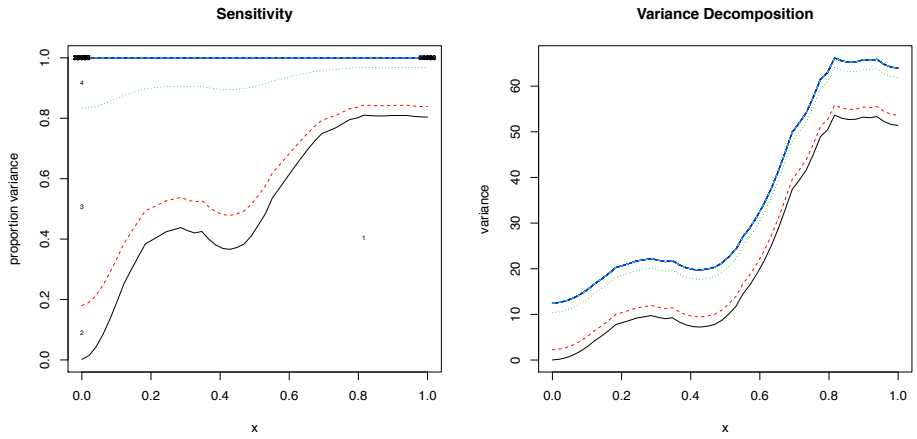

Figure 15: Functional sensitivity analysis - Friedman function with functional response.

```
R> dd <- read.table('https://archive.ics.uci.edu/ml/
+   machine-learning-databases/00291/airfoil_self_noise.dat')
```

We set aside 200 input combinations to use for testing.

```
R> set.seed(0)
R> test <- sample(nrow(dd), size=150)
R> x <- dd[-test, 1:5]
R> y <- dd[-test, 6]
```

We fit a BASS model using tempering.

```
R> mod <- bass(x, y, nmcmc = 20000, nburn = 10000, thin = 10,
+              temp.ladder = 1.1^(0:5), verbose = F)
```

We can predict as we have before. However, this prediction is for the mean function.

```
R> x.test <- dd[test, 1:5]
R> y.test <- dd[test, 6]
R> pred <- predict(mod, x.test)
```

Now, if we are interested in predicting actual data rather than the mean function, we can incorporate uncertainty from our estimate of σ^2. The vector `mult` below is the multiplyer we would use to get 95% prediction intervals.

```
R> mult <- 1.96 * sqrt(mod$s2)
R> q1 <- apply(pred - mult, 2, quantile, probs = .025)
R> q2 <- apply(pred + mult, 2, quantile, probs = .975)
R> mean((q1 < y.test) & (q2 > y.test))

[1] 0.94
```

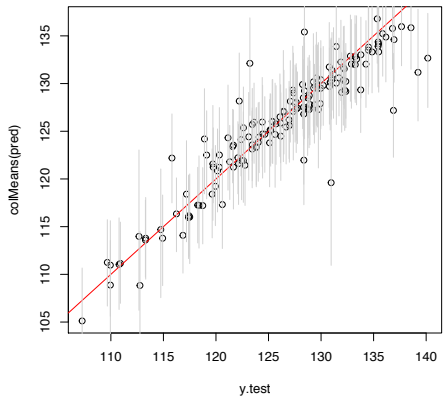

Figure 16: Prediction performance - air foil data.

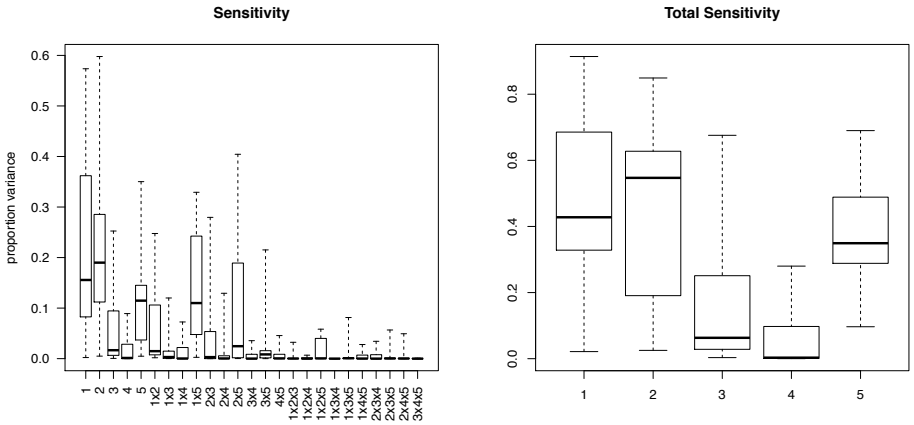

Figure 17: Sobol decomposition - air foil data.

This puts our empirical coverage where we would expect it to be. We can plot our 95% prediction intervals as follows, shown in Figure 16.

```
R> plot(y.test, colMeans(pred))
R> abline(a = 0, b = 1, col = 2)
R> segments(y.test, q1, y.test, q2, col = 'lightgrey')
```

Next, we can obtain and plot the Sobol' decomposition, shown in Figure 17.

```
R> sens <- sobol(mod, verbose = F)
R> plot(sens)
```

The uncertainty in the sensitivity indices in Figure 17 is significant and helps us to understand that there are many possible models for these data that use different variables and interactions. The proper characterization of this uncertainty would be impossible if our RJMCMC chain

was stuck in a mode. Hence, tempering is important in this problem. By exploring the posterior modes, tempering allows us to find not just a model that predicts well, but all the models that predict well.

4.6. Pollutant Spill Model

The final example we present is an emulation problem. The simulator is for modeling a pollutant spill caused by a chemical accident, obtained from Surjanovic and Bingham (2017). While fast to evaluate, this simulator provides a good testbed for BASS methods. The simulator has four inputs: (1) Mass of pollutant spilled at each of two locations (range $7-13$), (2) diffusion rate in the channel ($0.02-0.12$), (3) location of the second spill ($0.01-3$) and (4) time of the second spill ($30.01-30.295$). The simulator outputs a function in space (one dimension) and time that is the concentration of the pollutant.

We generate 10000 combinations of the four simulator inputs uniformly from within their respective ranges.

```
R> set.seed(0)
R> n <- 10000
R> x <- cbind(runif(n, 7, 13), runif(n, .02, .12), runif(n, .01, 3),
+             runif(n, 30.01, 30.295))
```

We specify six points in space and 20 time points. The functional grid we will pass to the `bass` function will thus have two columns, called `x.func` below.

```
R> s <- c(0, 0.5, 1, 1.5, 2, 2.5)
R> t <- seq(.3, 60, length.out = 20)
R> x.func <- expand.grid(t, s)
```

We use the `environ` function available from http://www.sfu.ca/~ssurjano/Code/environr.html to generate realizations of the simulator. We will model the log of the simulator output.

```
R> out <- t(apply(x, 1, environ, s = s, t = t))
R> y <- log(out + .01)
```

With this amount of data, we are presented with an extremely large model space to search through. In addition, since the data are smooth (no random noise), BASS models will tend to allocate a very large number of basis functions to try to capture the smoothness. In order to compensate for the large model space and the smoothness, we need to set an extreme prior on the number of basis functions to have a managable model. We do this by increasing `h2` by many orders of magnitude. In this example, we set `h2 = 1e150`. This results in a prior for the number of basis functions with very heavy weight near zero. Because of the large amount of data, we still get hundreds of basis functions.

Using such an extreme prior makes our multimodal posterior more peaked, and more difficult to explore. We may need hundreds of chains running at different temperatures in order to get the temperatures close enough to eachother to allow for frequent swapping. Another possibility that does not require picking a temperature ladder is to instead run multiple cold chains (at the true posterior) and allow them to swap states. This is a version of parallel hierarchical sampling introduced in Rigat and Mira (2012) that can be easily implemented by setting `temp.ladder = rep(1, n.chains)`.

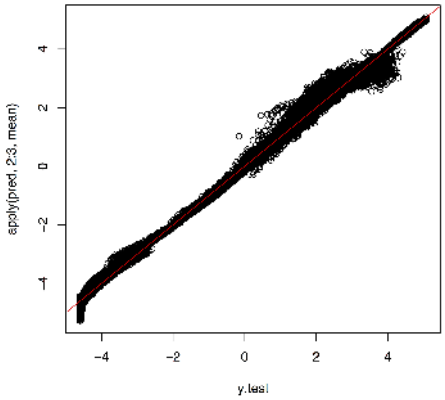

Figure 18: BASS prediction performance - pollutant spill model.

```
R> mod <- bass(x, y, xx.func = x.func, nmcmc = 110000, nburn = 100000,
+              thin = 10, h2 = 1e250, save.yhat = F, temp.ladder = rep(1, 10),
+              npart.func = 1, verbose = F, maxBasis = 175)
```

Note that we specify `save.yhat = F`. By default, the `bass` function saves in-sample predictions for all MCMC samples (post burn-in and thinned). This can be a sigificant storage burden when we have large amounts of functional data, as we do in this case. Changing the `save.yhat` parameter can relieve this. If in-sample predictions are of interest, they can be obtained after model fitting using the `predict` function.

As with the previous example, prediction here is for the mean function. Whatever error is left over (in σ^2) is inability of the BASS model to pick up high frequency signal.

```
R> n.test <- 1000
R> x.test <- cbind(runif(n.test, 7, 13),runif(n.test, .02, .12),
+                  runif(n.test, .01, 3), runif(n.test, 30.01, 30.295))
R> y.test <- log(t(apply(x.test, 1, environ, s = s, t = t)) + .01)
R> pred <- predict(mod, x.test)
```

A plot of the predicted (mean function) versus observed data is shown in Figure 18.

```
R> plot(y.test, apply(pred, 2:3, mean))
R> abline(a = 0, b = 1, col = 2)
```

To see what the predictions look like in space and time, consider the plots shown in Figure 19. These show posterior draws (in grey) of the mean function for one setting of the four inputs along with simulator output (in red).

```
R> pp <- pred[, 1, ]
R> ylim <- range(y)
R> par(mfrow=c(2, 3))
R> for(i in 1:length(s)) {
```

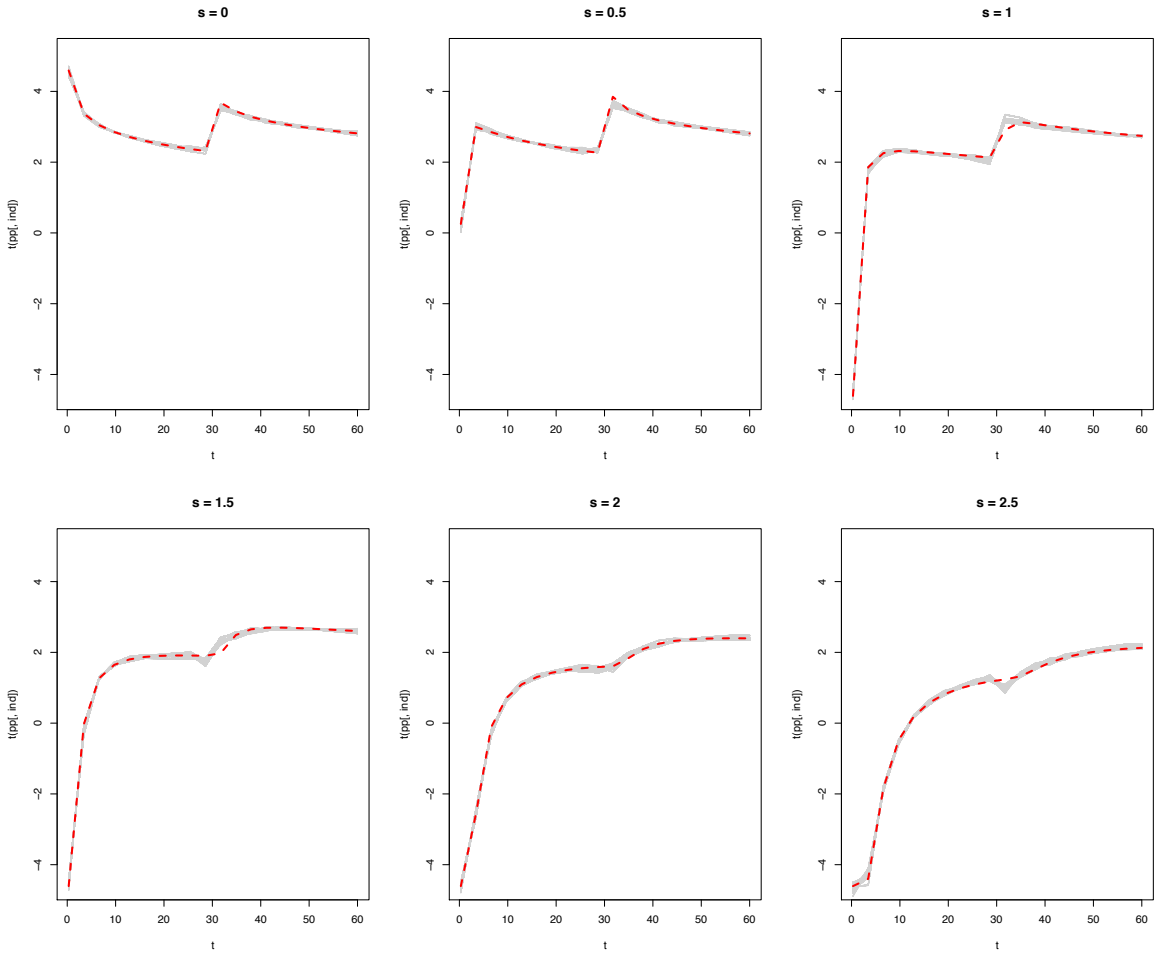

Figure 19: BASS prediction in space and time - pollutant spill model.

```
+   ind <- length(t) * (i - 1) + 1:length(t)
+   matplot(t, t(pp[, ind]), type = 'l', col = 'lightgrey',
+           ylim = ylim, main = paste('s =', s[i]))
+   lines(t, y.test[1, ind], col = 2, lwd = 2, lty = 2)
+ }
```

We can use the sensitivity analysis methods above, but we can get Sobol' indices as a function of either space or time. Below, we show how to get them as a function of time. We limit the models considered using `mcmc.use` to speed up computations. Since we have two functional inputs, we have two letters that can be included in these sensitivity plots (functional inputs are labeled with letters). Note that variable four is not included. This is because it did not explain any variance.

```
R> sens.func1 <- sobol(mod, mcmc.use = seq(1, 1000, 100),
+                      func.var = 1, xx.func.var = t, verbose = F)
R> plot(sens.func1)
```

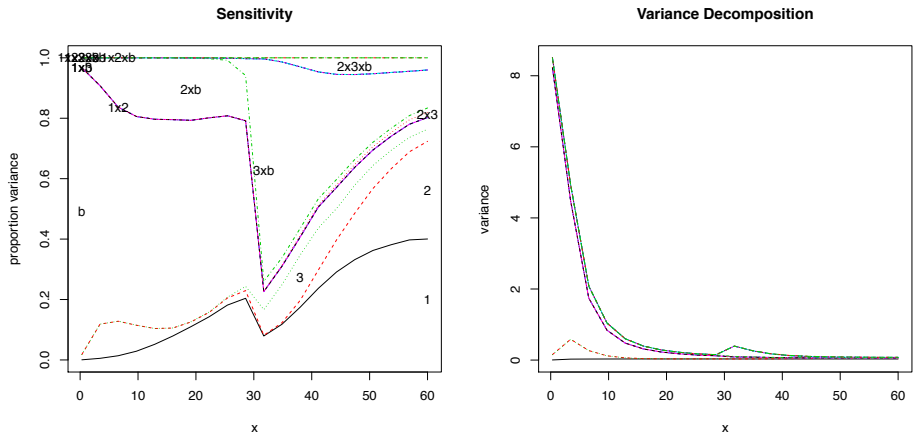

Figure 20: Sensitivity indices as a function of time - pollutant spill model.

5. Summary

Our proposed BASS framework provides a powerful general tool for nonparametric regression settings. It can be used for modeling with many continuous and categorical inputs, large sample size and functional response. It provides posterior sensitivity analyses without integration error. The MCMC approach to inference, especially using parallel tempering, yields posterior samples that can be used for probabilistic prediction. The **BASS** package makes these features accessible to users with minimal exposure. These capabilities have been demostrated with a set of examples involving different dimensions, categorical variables, functional responses, and large datasets.

References

Altekar G, Dwarkadas S, Huelsenbeck JP, Ronquist F (2004). "Parallel Metropolis Coupled Markov Chain Monte Carlo for Bayesian Phylogenetic Inference." *Bioinformatics*, **20**(3), 407–415.

Denison DG, Holmes CC, Mallick BK, Smith AF (2002). *Bayesian Methods for Nonlinear Classification and Regression*, volume 386. John Wiley & Sons.

Denison DG, Mallick BK, Smith AF (1998). "Bayesian MARS." *Statistics and Computing*, **8**(4), 337–346.

Fog A (2015). ***BiasedUrn**: Biased Urn Model Distributions*. R package version 1.07, URL https://CRAN.R-project.org/package=BiasedUrn.

Francom D (2016). ***BASS**: Bayesian Adaptive Spline Surfaces*. R package version 0.1.1, URL https://CRAN.R-project.org/package=BASS.

Francom D, Sansó B, Bulaevskaya V, Lucas D (2017). "Inferring Atmospheric Release Characteristics in a Large Computer Experiment using Bayesian Adaptive Splines." In preparation.

Francom D, Sansó B, Kupresanin A, Johannesson G (2016). "Sensitivity Analysis and Emulation for Functional Data using Bayesian Adaptive Splines." To appear in *Statistica Sinica*, URL https://www.soe.ucsc.edu/research/technical-reports.

Friedman JH (1991a). "Estimating Functions of Mixed Ordinal and Categorical Variables Using Adaptive Splines." *Technical report*, DTIC Document.

Friedman JH (1991b). "Multivariate Adaptive Regression Splines." *The Annals of Statistics*, pp. 1–67.

Gramacy R, Samworth R, King R (2010). "Importance Tempering." *Statistics and Computing*, **20**(1), 1–7.

Gramacy RB, Taddy M (2010). "Categorical Inputs, Sensitivity Analysis, Optimization and Importance Tempering with **tgp** Version 2, an R Package for Treed Gaussian Process Models." *Journal of Statistical Software*, **33**(6), 1–48.

Green PJ (1995). "Reversible jump Markov chain Monte Carlo computation and Bayesian model determination." *Biometrika*, **82**(4), 711–732.

Liang F, Paulo R, Molina G, Clyde MA, Berger JO (2008). "Mixtures of g Priors for Bayesian Variable Selection." *Journal of the American Statistical Association*, **103**(481).

Lichman M (2013). "UCI Machine Learning Repository." URL http://archive.ics.uci.edu/ml.

Milborrow S (2016). ***earth**: Multivariate Adaptive Regression Splines*. R package version 4.4.4, URL https://CRAN.R-project.org/package=earth.

Nott DJ, Kuk AY, Duc H (2005). "Efficient Sampling Schemes for Bayesian MARS Models with Many Predictors." *Statistics and Computing*, **15**(2), 93–101.

R Core Team (2016). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.

Rigat F, Mira A (2012). "Parallel Hierarchical Sampling: A General-Purpose Interacting Markov Chains Monte Carlo Algorithm." *Computational Statistics & Data Analysis*, **56**(6), 1450–1467.

Saltelli A, Ratto M, Andres T, Campolongo F, Cariboni J, Gatelli D, Saisana M, Tarantola S (2008). *Global Sensitivity Analysis: The Primer*. John Wiley & Sons.

Sobol' IM (2001). "Global Sensitivity Indices for Nonlinear Mathematical Models and Their Monte Carlo Estimates." *Mathematics and Computers in Simulation*, **55**(1), 271–280.

Surjanovic S, Bingham D (2017). "Virtual Library of Simulation Experiments: Test Functions and Datasets." Retrieved January 5, 2017, from http://www.sfu.ca/~ssurjano.

Affiliation:

Devin Francom
Department of Applied Mathematics and Statistics

University of California Santa Cruz
Santa Cruz, CA 95064
E-mail: dfrancom@ucsc.edu