FBC: Full Bayesian Calibration

Contents

Introduction

FBC is a package that uses the data from both physical experiment and computer simulation to calibrate the simulator model. Calibration is the process of fitting a statistical model to the observed data by adjusting calibration parameters. These parameters can represent various aspects of the simulation model, such as tuning hyperparameter of the model itself or unknown but fixed physical properties that are governed by physical system.

The FBC package is a based on the well-known Kennedy and O'Hagan (KOH) calibration model (2001). In its original formulation, KOH formulates a hierarchical Bayesian framework with two sequential phases. In the first phase, model hyperparameters are predicted using maximum likelihood estimation (MLE) method. In the second phase, Bayesian analysis is used to derive the posterior distribution of calibration parameters while fixing the hyperparameters found in the first phase. Neither KOH's original formulation nor later suggestions are fully Bayesian as they use MLE methods to estimate hyperparameters, largely due to computational infeasibility (CITATION: Higdon, Berger, other).

On the other hand, FBC runs a fully Bayesian model that includes both calibration parameters and model hyperparameters in its Bayesian framework. Implementation of the package optimizes the calibration process and memory management to increase computational efficiency. Moreover, the fully implemented Bayesian framework, enables the user to input the information about all parameters and hyperparameters in the form of prior specification. Common prior distribution are implemented in FBC to allows for high degree of flexibility in specifying the expert knowledge or the lack thereof. FBC runs a Markov Chain Monte Carlo (MCMC) algorithm to find the posterior distribution of calibration parameters and model hyperparameters. Furthermore, FBC can be used for prediction of mean response and model discrepancy for a new test input configuration.

(TODO: edit this paragraph to reflect the structure of the vignette in the end) The current vignette is structured into following sections: The first section (FBC Usage) explains the package functionality with a simple pedagogic example. Also in this section, the notation for inputs and outputs of both computer and physical experiments are introduced. The second section (Calibration Model) generalizes the example

introduced in the first section to build model components. Using the general notation while referencing the example, we describe the KOH calibration model and its modelling choices. The third section (Parameter Estimation), presents the theoretical results that characterize the posterior distribution of model parameters and MCMC-based estimation of parameters. The fourth section (Prediction), provides the results to derive MCMC-based predictions for new input configuration using the estimated parameters and MCMC samples of model components. The fifth section (Implementation) explains some of the implementation features and choices that distinguishes `FBC` from other implementations while justifying those choices. Finally, the last section (Application) provides two real-world and well-cited examples to demonstrate the full functionality and limitations of the `FBC` package. In the end, an appendix is provided to provide further details and a reference page to link for further readings.

1. `FBC` Usage

`FBC` package has two main public functions: `calibrate()` and `predict()`. As the name suggests, `calibrate()` takes the field and simulation data to calibrate the simulator model and `predict` takes a new input configuration and the calibration model (`fbc` object) to predict the field response at new input configuration and quantifies the uncertainty in prediction. In addition, `FBC` has a few helper function to aid in arguments entry and visualization.

1.1 Setup

Building a calibration model requires data from field experiment and computer simulation. To focus on the functionality of the package, we use a simple pedagogic example as experimental setting. In this setting, a wiffle ball is dropped from different heights and the time it takes to hit the ground is measured. This experiment has one experimental input height (h) and one calibration input gravity (g) to produce the response, time (t). Note that in the field experiment, the earth gravity is fixed but unknown to the experimenter (at least to some level of uncertainty) and therefore it is not part of the input data. The field experiment with aforementioned specification has been performed by Derek Bingham and Jason Loeppky (CITATION). The data field is loaded in the package environment under `ballField` name. To increase robustness, `calibrate()` requires the training field data to include both field response vector $\mathbf{t}$ and input matrix $\mathbf{h}$ (a vector in ball example) in a single input matrix `ballField`, where first column always contain the response, followed by experimental input variables as columns in the field data matrix.

```
library(FBC)
head(ballField, 3)
>         t     h
> [1,] 0.27 0.178
> [2,] 0.22 0.356
> [3,] 0.27 0.534
dim(ballField)
> [1] 63  2
```

Simulation of the ball drop experiment is easy to model using introductory physics results:

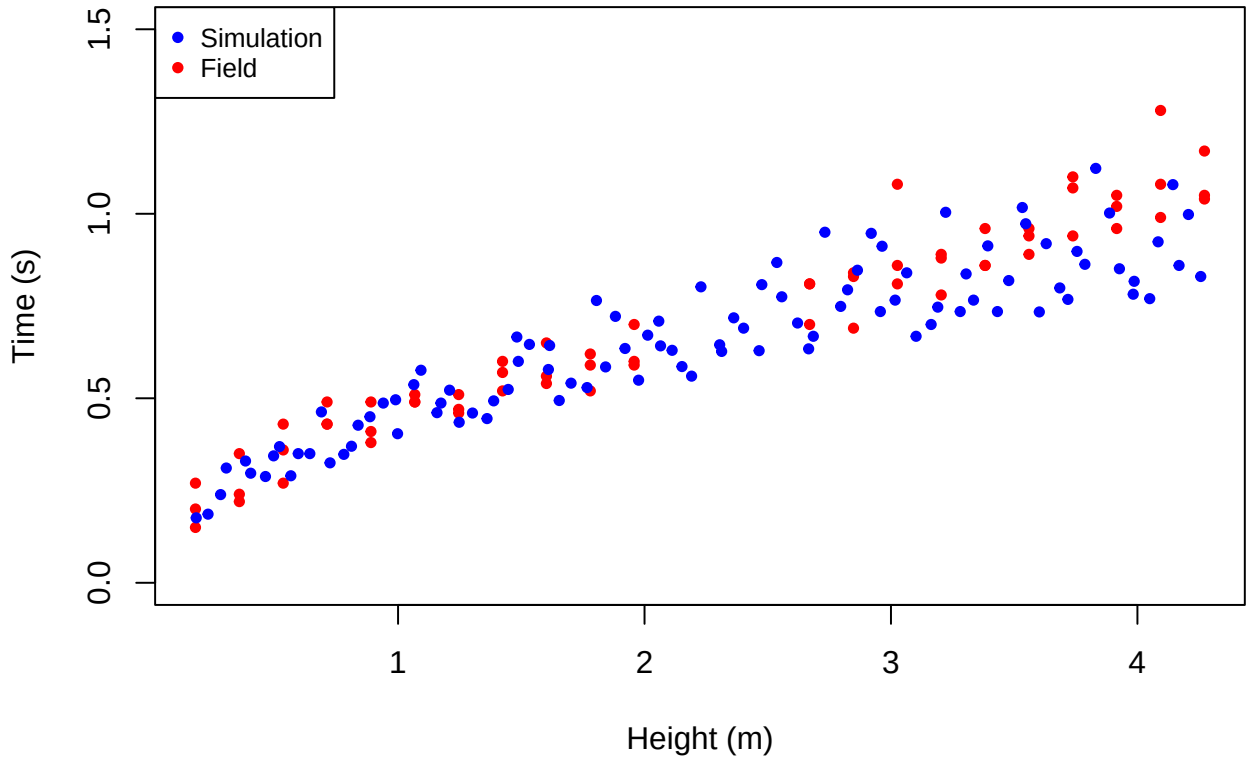
$$t = \sqrt{\frac{2h}{g}}$$

We have implemented the above mathematical model as a code that takes h and g as experimental and calibration inputs and returns t as simulator response. Latin Hypercube Design (CITATION) is used to create the input design matrix consisting of two columns: h and g. Similar to the field data, `calibrate` requires the simulation data to be packaged into a single matrix `ballSim`, where first column always represents response vector ($\mathbf{t}$), followed by experimental and calibration inputs as columns [$\mathbf{h}$ $\mathbf{g}$].

```
head(ballSim, 3)
>          t     h      g
> [1,] 0.404 0.998 12.220
> [2,] 0.487 0.940  7.909
> [3,] 0.450 0.886  8.736
dim(ballSim)
> [1] 100   3
```

Figure 1 shows the distribution of time versus height for both physical and simulation experiments. Note that for higher height values, the simulation responses (blue) underestimate their corresponding field response (red), displaying a systemic bias for simulation model.

Figure 1: Ball Drop Experiments



The ball example is a suitable pedagogical example as it has only one experimental input **h**) and one calibration input **g** to model response **t**. Using this toy example, we demonstrate the functionality of the package without getting to the details of a complex mathematical model. More complex and real-world examples are covered in the last section.

1.2 `calibrate()`

Arguments: The `calibrate()` function takes three sets of arguments from user: data, MCMC, and prior specification parameters. Other than data arguments which must be supplied by user, all other arguments have reasonable default values for ball example.

```
output <- calibrate(sim = ballSim, field = ballField,                              # Data
                    Nmcmc = 11000, nBurn = 1000, thinning = 50,                    # MCMC
                    kappaDist = "beta", kappaInit = NA, kappaP1 = 1.1, kappaP2 = 1.1, # Priors
```

```
                    hypers = set_hyperPriors(),
                    showProgress = FALSE)
```

The first and second arguments `sim` and `field` must be supplied by user. Both must be either in matrix or dataframe format representing the simulation and field data respectively. In both `ballSim` and `ballField` of our ball example, the first column represent the response **t** and the second column represent experimental input **h**. Additionally, `ballSim` has a third column that represents calibration input **g**. Calibration inputs are implicit in field experiment and are absent in the data matrix `ballField`.

$$\begin{array}{lcccc} & & \text{Response} & \text{Experimental} & \text{Calibration} \\ \texttt{ballSim} & = & \left[\quad \mathbf{t} \quad \right. & \mathbf{h} & \left. \quad \mathbf{g} \quad \right] \\ \texttt{ballField} & = & \left[\quad \mathbf{t} \quad \right. & \mathbf{h} & \left. \quad \right] \end{array}$$

The second set of arguments consists of MCMC parameters such as number of total iterations `Nmcmc`, number of burn-in iterations to be removed from the beginning of the chain `nBurn`, and `thinning`, which indicate the sampling rate to remove the autocorrelation from the sampled draws. For example, when `thinning = 50`, for every 50 draws from the result only one will be kept in order to remove the correlation between draws.

The third and last set of arguments consists of prior specification for each parameter of the model. The main goal of calibration is to estimate the calibration parameters (g in ball example). However, employing KOH calibration model introduces seven additional classes of hyperparameters to the model. All eight classes of parameters are unknown and must be estimated. As `FBC` employs a full Bayesian framework, the priors for all model parameters must be specified in advance. Throughout the package implementation and current guide a consistent notation is used to denote parameters: κ (calibration parameters), $\theta_{\mathbf{s}}$ (simulator correlation scale parameters), $\alpha_{\mathbf{s}}$ (simulator correlation smoothness parameters), $\theta_{\mathbf{b}}$ (bias-correction correlation scale parameters), $\alpha_{\mathbf{b}}$ (bias-correction correlation smoothness parameters), σ_s^2 (marginal simulator variance), σ_b^2 (marginal bias-correction variance), σ_ϵ^2 (measurement error variance), and μ_b (mean of bias-correction Gaussian Process. All calibration model components along with the introduced parameters are explained in more detail in next section.

For each class of parameters, there are four associated arguments. 1) distribution (suffixed with "dist" in argument names) type which is a character string determining the prior distribution. Currently, `FBC` supports almost all common distributions and can be chosen from ("uniform", "gaussian", "gamma", "beta", "lognormal", "logistic", "betashift", "exponential", "inversegamma", "jeffreys", "fixed"). "betashift" refers to a beta distribution that is shifted one unit to right to cover [1 2] interval and it is used to specify the priors for correlation smoothness parameters which must be constrained to [1 2] interval. Moreover, choosing "fixed" as distribution will exclude that class of parameters from the MCMC sampling. In this case, the given initial value will be used as fixed parameter value and p1 and p2 arguments are not used. 2) Initial value (suffixed with "init" in argument names) which represent starting point of the parameter(s) in MCMC algorithm. 3) and 4) represent the two parameters (p1 and p2) of the chosen distribution. For example, if "gaussian" distribution is used, p1 and p2 represent mean and variance of the distribution and if "uniform" distribution is used, p1 and p2 represent lower and upper bound of the distribution. Please note that not all distribution types require two arguments. In particular, "exponential" distribution only requires tone parameter (p1) and "jeffreys" requires none. In these cases the unused arguments are ignored..

From the classes of parameters mentioned, the prior for calibration parameters must be specified by user based on field knowledge as these parameters are problem-specific, even though a general vague prior is specified as default values. In contrast, all other parameters have reasonable default values based on literature and can be left unchanged. For this reason and to avoid unwanted complexity, all other priors are specified using a helper function `set_hyperPriors()`. Using this function without argument (which is the default value of the `hyper` argument) will specify all parameters based on their default values. However, expert knowledge about one or more of these parameters can be supplied using arguments of the `set_hyperPriors()` function

to change the default prior specifications. The following table summarizes all prior arguments to build the calibration model.

Table 1: *Argument names to specify priors for each parameter class.*

Parameter Class	Distribution	Initial Value	Shape Parameters
True field calibration inputs (κ)	`kappaDist`	`kappaInit`	`kappaP1` & `kappaP2`
Simulation GP scale (θ_s)	`thetaSDist`	`thetaSInit`	`thetaSP1` & `thetaSP2`
Simulation GP smoothness (α_s)	`alphaSDist`	`alphaSInit`	`alphaSP1` & `alphaSP2`
Bias-correction GP scale (θ_b)	`thetaBDist`	`thetaBInit`	`thetaBP1` & `thetaBP2`
Bias-correction GP smoothness (α_b)	`alphaBDist`	`alphaBInit`	`alphaBP1` &`alphaBP2`
Simulation marginal variance (σ_s^2)	`sigma2SDist`	`sigma2SInit`	`sigma2SP1` & `sigma2SP2`
Bias-correction marginal variance (σ_b^2)	`sigma2BDist`	`sigma2BInit`	`sigma2BP1` & `sigma2BP2`
Measurement error variance (σ_ϵ^2)	`sigma2EDist`	`sigma2EInit`	`sigma2EP1` & `sigma2EP2`
Bias-correction mean (μ_b)	`muBDist`	`muBInit`	`muBP1` & `muBP2`

Distribution arguments must be chosen from the string list of implemented prior distributions and initial value and the two shape parameter arguments must be numeric. Note that some classes of parameters may contain more than one parameters. In this case, the argument values can be vector instead of default scaler. In this case all four fields of that parameter class must be in vector format with same length as number of parameters in the same class. For example if there are five calibration parameters, `kappaDist` can either be a scaler, in which case the distribution for all calibration parameters will be set to that scaler value (and `kappaInit`, `kappaP1`, and `kappaP2` must be also scaler), or a vector of length five that supplies the distribution types for all calibration parameters (and `kappaInit`, `kappaP1`, and `kappaP2` must also be vector of length 5).

Moreover, there is a logical argument `showProgress` that indicate whether function must show the progress in calibration on console. This will put the `calibrate()` in interactive mode and will show the percentage of the MCMC draws along with sample draws.

Output: The output of the `calibrate()` function is a `fbc` object that contains the samples from posterior joint distribution of parameters, along with other model information.

```
names(output)
>  [1] "Phi"        "estimates"  "logPost"    "priors"     "acceptance" "vars"       "data"
>  [8] "scale"      "indices"    "priorFns"   "proposalSD"
```

The first and main component of the output is matrix `Phi` whose columns represent the sample of posterior densities for each unknown parameter of the model in the same order as parameter classes in table 1. Since κ, θ_s, α_s, θ_b, α_b parameter classes may contain more than one parameter, they are suffixed by a number that represents the index of the parameter in the class. For example, if there are 3 calibration inputs, the first p columns of the matrix `Phi` represent posterior density of calibration parameters and the column headers

will be `kappa1`, `kappa2`, and `kappa3`. In the ball example, there is only one calibration parameter g, which is denoted by κ_1 and represented by `kappa1` in matrix `Phi`. Each row of the matrix `Phi` represents a MCMC draw.

```
head(output$Phi, 3)
>   kappa1 thetaS1 thetaS2 alphaS1 alphaS2 thetaB1 alphaB1 sigma2S sigma2B sigma2E   muB
> 1   6.16    5.14    0.69    1.94    1.84    0.68    1.90    0.09    0.30    0.09 -0.35
> 2   8.72    4.92    1.04    1.94    1.97    0.17    1.81    0.09    0.66    0.09  0.24
> 3  13.70    4.35    0.76    1.90    1.93    0.43    1.79    0.10    0.44    0.08  0.15
```

All other columns of matrix `Phi` represent the samples for other model hyperparameters. Table 2 provides an overview of the model parameters and the notation to represent them in ball example. Later sections will explain in detail why are these hyperparameters introduced, what do they represent, and how they are estimated.

Table 2: Notation used in matrix `Phi` to represent parameters in ball example.

Column	Notation	Description
`kappa1`	κ_1	Unknown value of true calibration input g
`thetaS1`	θ_{s1}	Scale parameter of h input for simulator correlation
`thetaS2`	θ_{s2}	Scale parameter of g input for simulator correlation
`alphaS1`	α_{s1}	Smoothness parameter of h input for simulator correlation
`alphaS2`	α_{s2}	Smoothness parameter of g input for simulator correlation
`thetaB1`	θ_{b1}	Scale parameter of h input for bias-correction correlation
`alphaB1`	α_{b1}	Smoothness parameter of h input for bias-correction correlation
`sigma2S`	σ_s^2	Marginal variance of simulator covariance
`sigma2B`	σ_b^2	Marginal variance of bias-correction covariance
`sigma2E`	σ_ϵ^2	Variance of random measurement error in field
`muB`	μ_b	bias-correction mean

`estimates` is a data frame that provides a summary table of all model parameters. `summary()` function also provides the same statistics given a `fbc` object. `logPost` is vector of computed posterior log likelihood given a parameter draw from `Phi` matrix. Therefore, the length of the `logPost` vector is same as number of rows in `Phi`. `priors` is a nested list that contains prior specifications for all parameters (including calibration parameters). `acceptance` is a vector of same length as number of unknown parameters and represent the acceptance rate of each parameter in MCMC algorithm. The current implementation of MCMC algorithm employs Metropolis-Within-Gibbs variation, which is a one-dimensional proposal scheme and the optimal acceptance rate must be close to 0.44. `vars` is a vector of character string with the same length as number of parameters and contains the parameter notation used in code and as column headers. `data` is a list consisting of training data in the form of vectors and matrices. `scale` is a numeric vector that contains scaling factors used during calibration to scale the training data. `indices` is a list of vectors containing the indices of parameters in each row of `Phi` matrix. `priorFns` is a list of prior functions that are created during calibration based on given prior specifications. And finally, `proposalSD` is a numeric vector that represents the final standard deviation of proposal for each parameter.

1.3 predict()

Arguments: Similar to any other predict function, `predict()` requires a model object argument called `object`, which in `FBC` package must be a `fbc` object, along with an argument representing a new input configuration called `newdata`. Moreover, current implementation of the `predict()`, support to different methods of prediction: Maximum A Posteriori ("MAP") and MCMC-based fully Bayesian ("Bayesian") methods. The method can be selected using `method` argument that can take a character string value of either "MAP" or "Bayesian".

```
predsMAP   <- predict(object = output, newdata = matrix(c(2.2, 2.4), ncol = 1), method = "MAP")
predsBayes <- predict(object = output, newdata = matrix(c(2.2, 2.4), ncol = 1), method = "Bayesian")
```

Output: `predict()` returns a list consisting of two fields: `pred`, which is a vector of the predicted response ($\hat{t}$ in ball example) with same length as `newdata` rows, and `se`, which is a vector of the predicted response's standard errors, again with the same length as `newdata` rows.

```
predsMAP
> $pred
> [1] 0.6922523 0.7301342
>
> $se
> [1] 0.077 0.077
predsBayes
> $pred
> [1] 0.695 0.733
>
> $se
> [1] 0.071 0.071
```

In "Bayesian" method, which is the default method, MCMC draws of calibration model parameters are used to form a distribution of each predicted value. In particular, each rows of matrix `Phi` from the output of the calibration model, is used to predict the response for a single row (a new input configuration) of `newdata`. Therefore, for each observation a MCMC-based distribution of predicted responses are computed. Then, the predictive mean and variance of this distribution are used to computed point predictions as well as standard errors for predictions.

In "MAP" method, the row of `Phi` matrix that results in maximum log posterior, is extracted and taken as model parameters to compute both point estimates and standard errors. This method is much faster than the "Bayesian" method as it only computes the prediction for each row of observation once. More detail on theory behind prediction methods can be found in section

1.4 set_hyperPriors():

As mentioned hyperparameters of the calibration model can be set using this function. All hyperparameters have reasonable default values. Therefore `set_hyperParameters()` can be used without arguments to set the hyperparameters. In fact, the default value of the `hyper` argument in `calibrate()` is the function `set_hyperParameters()` without any argument. Nevertheless, when there is prior belief about structure of correlation structures (either simulator GP or bias-correction GP), these beliefs can be applied to the model in the form of prior specification using `set_hyperPriors()`

```
priors <- set_hyperPriors(thetaSDist = "beta", thetaBP2 = 6)
```

1.5 `summary()` and `print()`:

Both of these generic functions are implemented to work with the output of `calibrate()` function. In particular, given a `fbc` object, `summary()` displays the summary data frame of calibration parameters which is the `estimate` component of `calibrate()` output. Similarly, `print()` will print the data frame in the console.

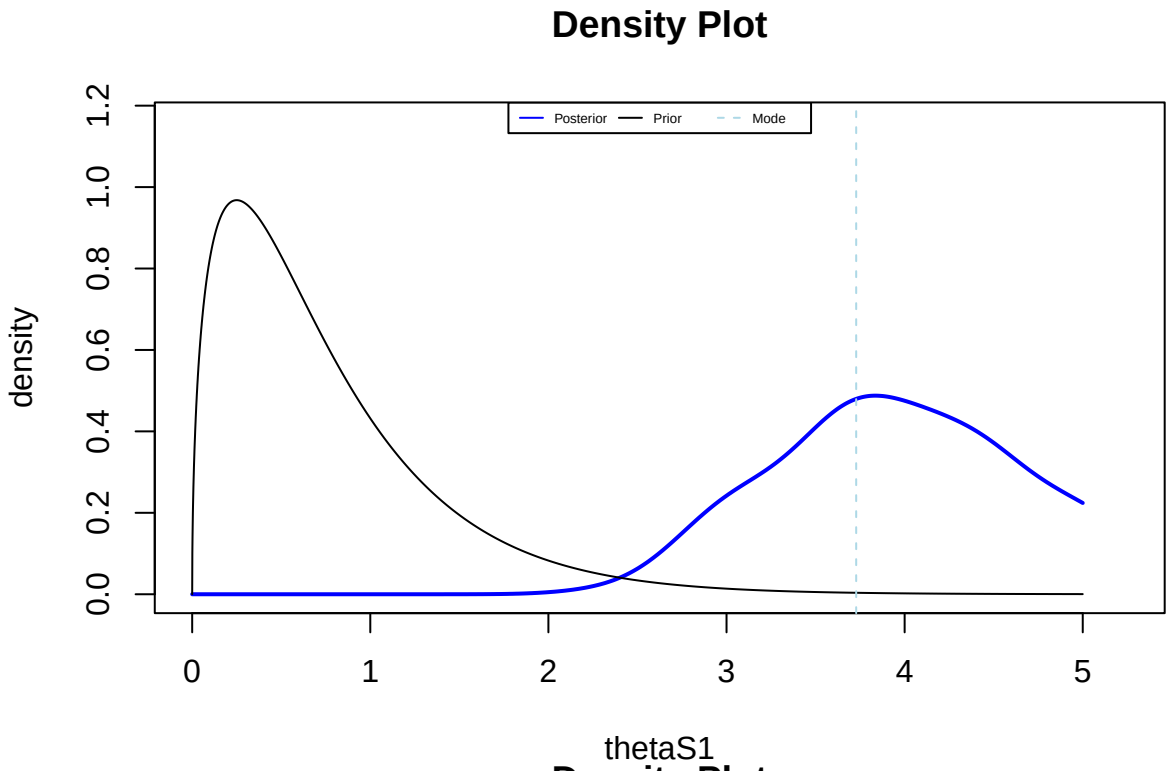
```
summary(output) # print(output) has exactly the same output
>           mean median   mode  lwr50  upr50  lwr80  upr80    sd
> kappa1   9.118  8.594  7.221  7.521 10.233  6.914 12.316 1.995
> thetaS1  4.120  4.044  3.728  3.545  4.576  3.049  5.223 0.837
> thetaS2  0.741  0.710  0.610  0.599  0.839  0.510  1.017 0.210
> alphaS1  1.896  1.906  1.921  1.865  1.927  1.837  1.946 0.046
> alphaS2  1.862  1.862  1.817  1.817  1.911  1.780  1.946 0.065
> thetaB1  0.601  0.520  0.196  0.287  0.791  0.170  1.190 0.418
> alphaB1  1.793  1.824  1.772  1.719  1.905  1.572  1.961 0.147
> sigma2S  0.091  0.089  0.082  0.082  0.098  0.076  0.107 0.012
> sigma2B  0.785  0.611  0.280  0.354  0.984  0.267  1.545 0.627
> sigma2E  0.100  0.100  0.103  0.089  0.109  0.081  0.120 0.016
> muB     -0.119 -0.118 -0.140 -0.257  0.021 -0.349  0.136 0.192
```

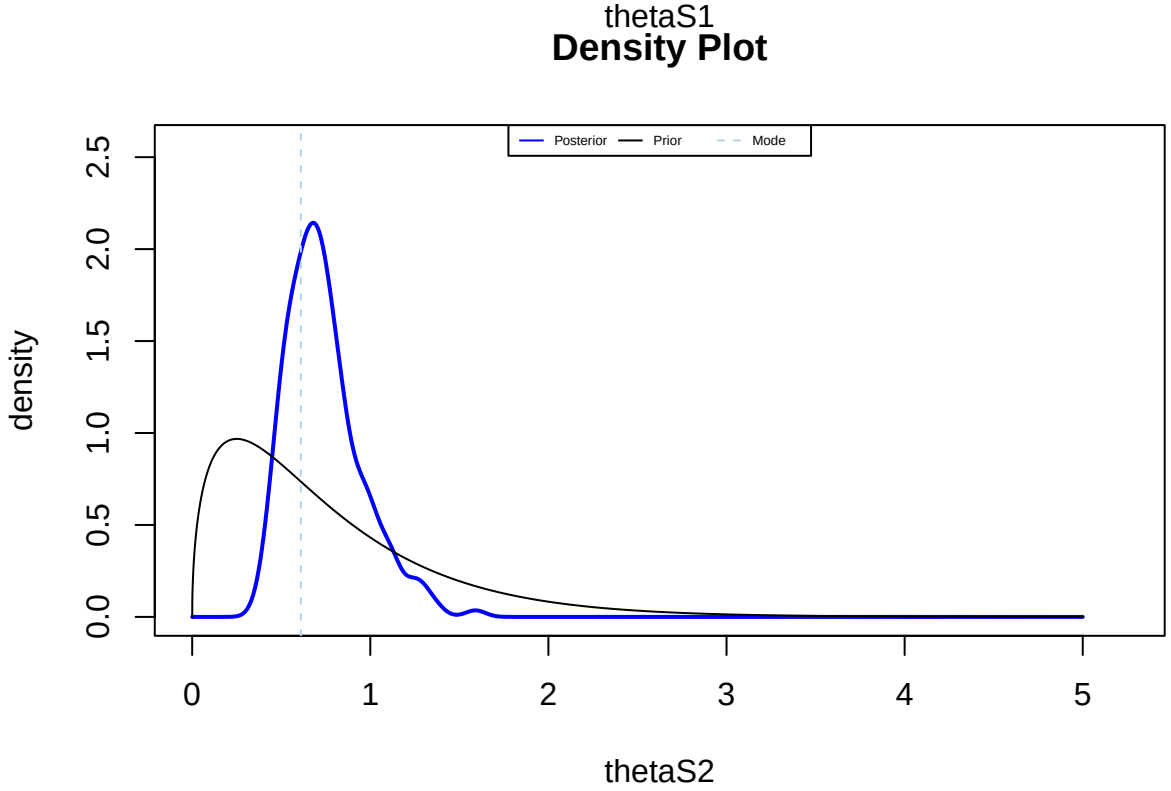
1.6 `plot()`

As name suggests, the implementation of this generic function enables visualization of a calibration model results. Given a calibration model in the form of a `fbc` object, `plot()` can visualize the model in three different mode, which can be chosen by supplying the `type` argument. In particular, `type` must be a character string form "density", "trace", and "fits".

Using "density", which is the default value of the `type` argument, `plot()` will plot the marginal posterior density distribution for the given parameter. It does so by estimating a density function given the MCMC-based posterior draws of the parameter. It also plots the prior distribution of the given parameter in the same plot for ease of comparison and visualizes the empirical mode of the posterior density. The given parameter must also be given as character string sing the `parameter` argument. The parameter name is consistent with the notation used throughout the package and current vignette and must be chosen from "kappa", "thetaS", "alphaS", "thetaB", "alphaB", "sigma2S", "sigma2B","sigma2E", or "muB". The default value for `parameter` argument is "kappa" (calibration parameters), which usually is the parameter of the interest. Note that `parameter` argument characterizes the class of parameters and when there is more than one parameter in that class,`plot()` will plot the density distribution for all parameters in that class in separate plots.

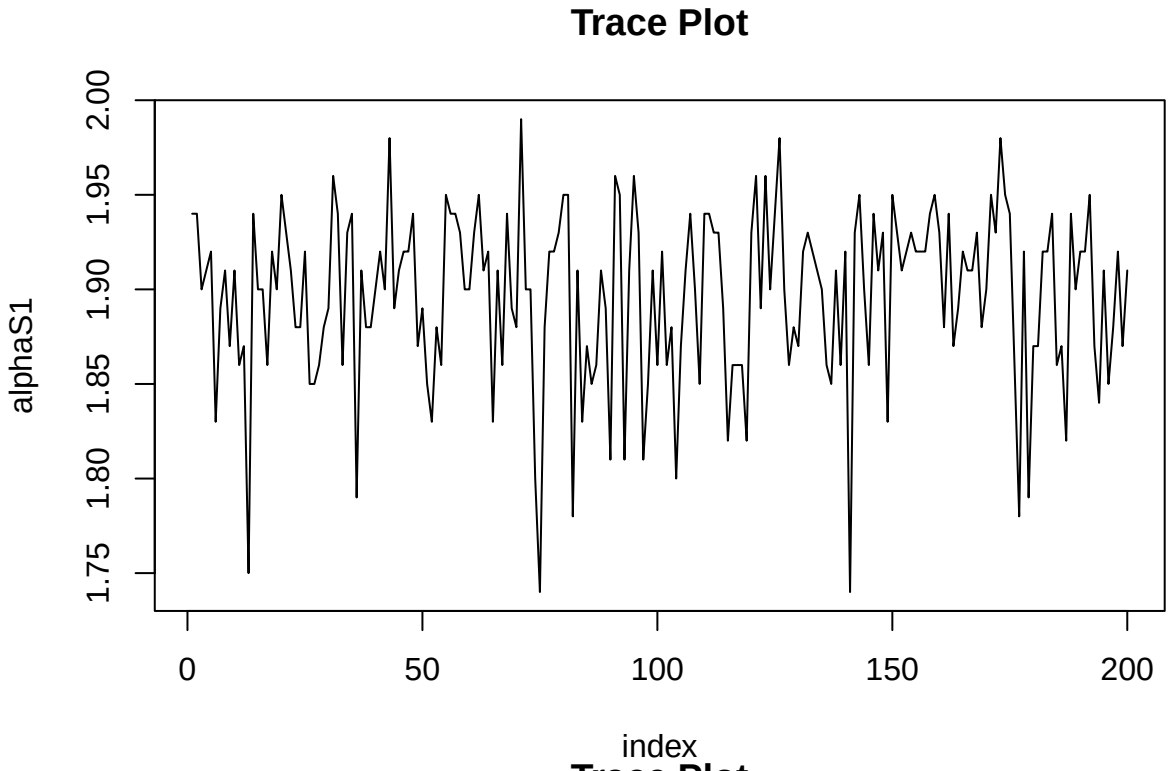
```
# plot other model parameters (note that since there are two correlation scale parameters for
# simulator GP, there will be two plots)
plot(output, parameter = "thetaS")
```

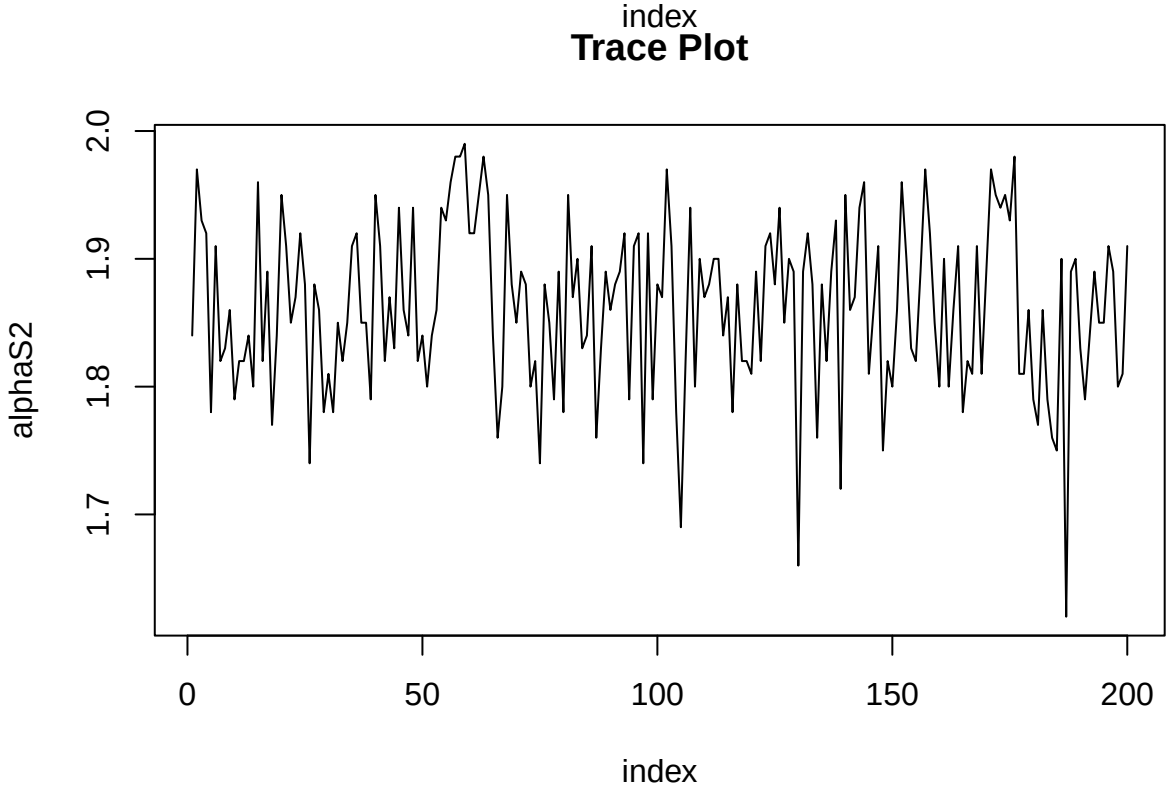




Using "trace" as `type` argument, `plot()` will plot the progression of the given parameter as MCMC draws are taken. It is similar to the time series of the parameter but indexed with number of iteration in MCMC rather than time. The trace plot can be used to determine whether there is good mixing in MCMC draws. Using "trace" as `type` argument also requires supplying the `parameter` from aforementioned list of possible parameter classes. And similar to density plots, trace plots will be plotted for all of the parameters in the given class in separate plots.

```r
# plot the trace of MCMC draws (note that this plot is useful to determine convergence and
# stationarity of the parameter draws. This minimal example dhas only few draws!)
plot(output, parameter = "alphaS", type = "trace")
```

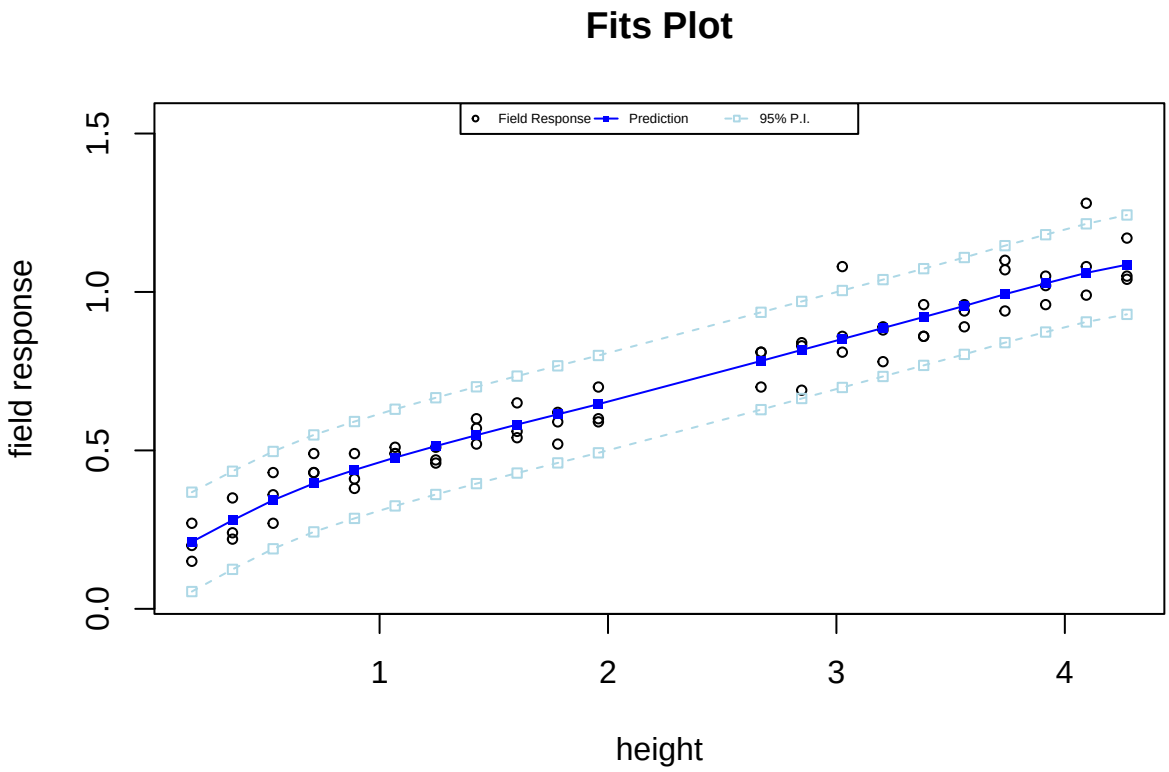




And finally using "fits" as `type` argument, `plot()` will plot the fitted values of the response versus all experimental variables in separate plots. The `parameter` argument is not required for this type and will

be ignored. Fits plots can be used to visually determine the goodness of fits versus actual response in interpolations. Internally, `plot()` will use `predict()` function (using "MAP" method) to compute the fitted values for the training input configurations and will plot them in conjunct with actual values. Furthermore, `xlab` argument can be supplied with a character string to characterize the experimental variables' names.

```
# plot the fitted values of the calibration model. It will plot the fitted values versus all
# experimental inputs, in this example only one!
plot(output, type = "fits", xlab = "height")
```



There are also three more exported functions in `FBC` package that are not required to build calibration models or to predict based on calibration models. These functions are used internally in the package but since they offer functionalists that are not supported in base R, they are exported for use.

1.7 `correlation()`:

This function is used to compute the correlation between rows of two given matrices assuming a power exponential correlation family structure. This family is a generalization of correlation where distinct scale and smoothness hyperparameters are used for different dimensions of the given data (columns of given matrices) when computing the correlation. A more thorough definition of power exponential correlation family is explained in section two. `correlation()` function is more general than R's base correlation function `cor()`, as it treats different dimensions using separate scale parameters and also allows for use of smoothness parameters (also for different dimensions) when computing the correlation. Note that only the first matrix, characterized by argument `X` must be supplied and the default value for the second matrix, characterized by `Y` is `NULL`. When only a single matrix is supplied, `correlation()` will compute the correlation of that matrix with itself. Other than the given matrix or matrices, which must have same number of columns, user must supply two vectors with `theta` and `alpha` arguments to characterize scale and smoothness parameters. The length of both vectors either must be same and equal to the number of columns in given matrices, or they must be scaler in which case for all dimensions same values of scale or smoothness will be used.

```
X     <- matrix(c(1, 3, 5,
                  2, 2, 6,
                  1, 4, 1), nrow = 3, byrow = TRUE)

Y     <- matrix(c(7, 3, 0,
                  2, 2, 4), nrow = 2, byrow = TRUE)

sc    <- c(1, 2, 3) # scale parameters of correlation structure
sm    <- c(2, 1, 2) # smoothness parameters of correlation structure

# correlation of a matrix with itself
round(correlation(X, theta = sc, alpha = sm), 5)
>          [,1]    [,2] [,3]
> [1,] 1.00000 0.00248    0
> [2,] 0.00248 1.00000    0
> [3,] 0.00000 0.00000    1

# correlation between two matrices
round(correlation(X, Y, theta = sc, alpha = sm), 5)
>      [,1]    [,2]
> [1,]    0 0.00248
> [2,]    0 0.00001
> [3,]    0 0.00000
```

1.6 prior_builder()

This function will create a prior function based on given distribution and distribution parameters. The function has three arguments: `prior` takes a character string from ("uniform", "gaussian", "gamma", "beta", "lognormal", "logistic", "betashift", "exponential", "inversegamma", "jeffreys", "fixed") and characterizes the type of distribution. When `fixed` is used, the prior function is simply `x = 1`. The next two arguments, `p1` and `p2` characterize the parameters of distribution. For example for gamma distribution `p1` and `p2` determine shape and scale of the gamma distribution, or for Gaussian distribution, `p1` and `p2` determine mean and standard deviation of the Gaussian distribution. The output of `prior_builder()` is a function that given a value, computes the probability density of the chosen distribution. To reduce the load of internal computation, `prior_builder()` computes the log of density and if used externally must be transformed.

```
# create a prior function for beta(2, 5). Note that the function compute log of priors and must be tran
pr_fun <- prior_builder(prior = "beta", p1 = 2, p2 = 5)
round(exp(pr_fun(c(-1, 0, 0.1, 0.5, 0.9, 1, 2))), 3)
> [1] 0.000 0.000 1.968 0.938 0.003 0.000 0.000

# create a prior function for a Uniform distribution with lower bound of -10, and upper bound of 10
pr_fun <- prior_builder(prior = "uniform", p1 = -10, p2 = 10)
round(exp(pr_fun(c(-11, -5, 0, 4, 10, 12))), 3)
> [1] 0.00 0.05 0.05 0.05 0.05 0.00

# create a prior function for Gaussian distribution with mean of 1 and standard deviation of 2
pr_fun <- prior_builder(prior = "gaussian", p1 = 1, p2 = 2)
round(exp(pr_fun(c(-9, -5, -3, -1, 1, 3, 5, 7, 11))), 3)
> [1] 0.000 0.002 0.027 0.121 0.199 0.121 0.027 0.002 0.000
```

1.7 `pmode()`

This function computes an estimate of the mode of a continuous distribution. It works similar to a histogram, in which the domain of the distribution is broken into same length bins. For each bin, the draws of the distribution that fall within that bin is counted and at the end the mean of the bin with highest count is returned as estimated mode. The function also takes the number of bins through `breaks` argument. The default value is `NULL`, which makes `pmode` to determine the number of required bins dynamically based on number of data points and the domain of the distribution.

```
# find the estimated mode of a vector
vec <- runif(100, 0, 10)
pmode(vec)
> [1] 5.739326
pmode(vec, breaks = 10)
> [1] 2.457838
```

2. Calibration Model

Calibration model is statistical model that represent both field and simulator response as function of input configuration. In this section, we explain the theory behind building a calibration model and relate the notation used in the formulation of calibration model and in the package implementation to our ball example.

2.1 Data

A computer experiment or simulation is simply running a computer code at different input configurations and recording the response. The code is an implementation of the mathematical model that is intended to mimic the physical experiment. In general, a simulation has p experimental inputs but also has q additional calibration inputs that are either tuning parameters or unknown physical properties that are not controllable by field experimenter. Table 2 describes the response and inputs of a computer experiment with m observations using matrix notation. The subscript s is used to denote simulation.

Table 2: Notation used to represent simulation data component of calibration model.

Notation	***Description***	***Ball Example***
m	Number of simulation runs	100
p	Number of experimental inputs	1
q	Number of calibration inputs	1
$\mathbf{x_s}$	Simulation input vector containing $(p+q)$ inputs	(h, g) (vector of length 2)
y_s	Univariate simulation response	t (scaler)
$\mathbf{X_s}$	$(m \times (p+q))$ simulation input matrix	[**h g**] ((100×2) matrix)
$\mathbf{y_s}$	Vector of m univariate simulation response	**t** (vector of length 100)

On the other hand, a physical experiment consists of n observations of a physical property, each with p experimental inputs. The calibration inputs are implicit in physical experiment and their values are unknown

but assumed to be fixed throughout observations. To represent both experiments in a unified structure, the unknown calibration inputs of field experiment κ must be augmented to experimental inputs, so that both physical and computer experiments have $(p+q)$ inputs (h and g in ball example) and a univariate response (t in ball example). Assuming vector $\mathbf{x_f}$ represent experimental input, the augmented input vector is denoted by $\mathbf{x}_\kappa$. Since $\mathbf{x_f}$ has p elements and κ has q elements, both will have $(p+q)$ elements similar to $\mathbf{x_s}$. Stacking the input vectors, we can represent all field and augmented input vectors using $\mathbf{X_f}$ and $\mathbf{X}_\kappa$. Similarly, vector $\mathbf{y_f}$ represents all field observations. Subscripts f and $\kappa$are used to denote field and augmented data respectively.

Elements of vector κ are parameters of the calibration model and will be estimated by `calibrate()` [1].

[1] *The first q columns of matrix* `Phi` *in* `calibrate` *output represent the MCMC samples of posterior densities for calibration parameters. In our ball example, there is only one calibration parameter (gravity), which is denoted by* κ_1 *and represented by* `kappa1` *in* `Phi`*..*

Table 3: Notation used to represent field data component of calibration model.

Notation	***Description***	***Ball Example***
n	Number of field observations	63
p	Number of experimental inputs	1
$\mathbf{x_f}$	Field input vector containing p experimental inputs	h (scaler)[2]
κ	Vector of unknown true calibration inputs in field experiment	true value of gravity κ_1
$\mathbf{x}_\kappa$	Augmented field input vector containing $(p+q)$ inputs[3]	$(h, \kappa_1)^2$ (vector of length 2)
y_f	Univariate field response	t (scaler)
$\mathbf{X_f}$	$(n \times p)$ field input matrix	$\mathbf{h}$ (vector[2] of length 63)
$\mathbf{X}_\kappa$	$(n \times (p+q))$ augmented field input matrix	$[\mathbf{h} \quad \kappa_\mathbf{1}]$[4]
$\mathbf{y_f}$	Vector of n univariate field response	$\mathbf{t}$ (vector of length 63)

All of the data components in Table 2 and 3 are generated internally to build the calibration model. The user is only required to provide two matrices representing the field and simulation datasets in their entirety through `sim` and `field` arguments of `calibrate()`.

2.2 KOH Model

Random Functions: KOH models the functional relationship between simulation input and output as a realization of a random function $\eta(\mathbf{x_s})$. Similarly, KOH models the functional relationship between field input and output as a realization of random function $\eta(\mathbf{x}_\kappa)$ but acknowledges a systemic model discrepancy and measurement errors. As a result, KOH models the discrepancy by adding a bias-correction term as realization of another random function $\delta_\kappa(\mathbf{x_f})$. The error term ϵ is considered to be an independent draw from a normal distribution with zero mean and unknown variance σ^2_ϵ.

$$\epsilon \quad \sim \mathcal{N}(0, \sigma^2_\epsilon)$$

$$y_f = \eta(\mathbf{x}_\kappa) + \delta_\kappa(\mathbf{x_f}) + \epsilon$$

$$y_s = \eta(\mathbf{x_s})$$

Therefore other than κ and σ^2_ϵ parameters, the random functions $\eta(.)$ and $\delta_\kappa(.)$ are also unknown and must be specified. KOH models $\eta(.)$ and $\delta_\kappa(.)$ by two independent Gaussian Processes (GP).

[2] *In ball example there is only one experimental input and therefore* $\mathbf{x_f}$ *is a vector of length one or scaler. Similarly, in its matrix notation,* $\mathbf{X_f}$ *is a* $(n \times 1)$ *matrix or a vector of length* n*..*

[3] *Note that calibration parameters* κ *are often assumed to be unchanged throughout field experiment. Therefore, same* (κ) *vector is augmented to all of the field input configurations..*

[4] *Since calibration input is fixed for all field observation,* $\kappa_\mathbf{1} = (\kappa_\mathbf{1}, ..., \kappa_\mathbf{1})$ *vector of length 63}] (*(100×2) *matrix)..*

$$\eta(.) \sim GP\ (0,\ \sigma_s^2.R_s(.,.))$$
$$\delta_\kappa(.) \sim GP\ (0,\ \sigma_b^2.R_b(.,.))$$

Where σ_s^2 and σ_b^2 are marginal variance of simulator and bias-correction GPs, and $R_s(.,.))$ and $R_b(.,.))$ are correlation matrix of simulator GP (using full input matrix $\mathbf{X_s}$ or $\mathbf{X}_\kappa$) and bias-correction GP (using field input matrix X_f).

Note that means of GPs are considered to be zero because `calibrate()` function first standardizes simulation response $\mathbf{y_s}$ (mean zero and standard deviation of one) and then scales field response according to $\mathbf{y_s}$'s scaling factors. Furthermore, the simulator inputs (both experimental and calibration) are scaled to span $[0, 1]$ and the scaling factors of simulation experimental inputs are used to scale field experimental inputs. As a result considering zero mean for both processes seem reasonable. (TODO: look into the effect of constant mean for discrepancy GP).

Correlation Structure: `FBC` employs a power exponential correlation family to represent the correlation structure of both GPs. Assuming $\mathbf{x}$ and $\mathbf{x}'$ are two rows of full input matrix (either $\mathbf{X_s}$ or $\mathbf{X}_\kappa$) and $\mathbf{x_f}$ and $\mathbf{x_f'}$ are two rows of field experimental input matrix ($\mathbf{X_f}$), the correlation matrices $R_s(\mathbf{x}, \mathbf{x}')$ and $R_b(\mathbf{x_f}, \mathbf{x_f'})$ are defined as following:

$$R_s(\mathbf{x},\ \mathbf{x}') = \prod_{i=1}^{p+q} e^{-\theta_i |x_i - x_i'|^{\alpha_i}}$$

$$R_b(\mathbf{x_f},\ \mathbf{x_f}') = \prod_{j=1}^{p} e^{-\theta_j |x_j - x_j'|^{\alpha_j}}$$

Using separable power exponential correlation family introduces two new hyperparameters for each input: scale (θ_i) and smoothness (α_i). Together, they flexibly determine the shape of correlation structure. Table 4 introduce the notation used for hyperparameters of $\eta(.)$ and $\delta_\kappa(.)$.

Table 4: *New Parameters To Build KOH Calibration Model.*

GP	Hyperparameters	Ball Example (Columns of `Phi`)
$\eta(.)$	$(\theta_{s1}, ..., \theta_{s(p+q)}, \alpha_{s1}, ..., \alpha_{s(p+q)}, \sigma_s^2)$	(`thetaS1`, `thetaS2`,`alphaS1`,`alphaS2`, `sigma2S`)
$\delta_\kappa(.)$	$(\theta_{b1}, ..., \theta_{bp}, \alpha_{b1}, ..., \alpha_{bp}, \sigma_b^2)$	(`thetaB1`, `alphaB1`, `sigma2B`)

Full Model: After augmentation of true calibration inputs (vector κ to field data, both simulation and field experiment have the same input. KOH combines both components to build a joint model. The joint vector of all parameters in the final calibration model is denoted by:

$$\phi = (\kappa_1,\ ...,\ \kappa_q,\ \theta_{s1},\ ...,\ \theta_{s(p+q)},\ \alpha_{s1},\ ...,\ \alpha_{s(p+q)},\ \theta_{b1},\ ...,\ \theta_{bp},\ \alpha_{b1},\ ...,\ \alpha_{bp},\ \sigma_s^2,\ \sigma_b^2,\ \sigma_\epsilon^2)$$

Note that in the ball example, the column headers of matrix `Phi` exactly match to model parameters.

```
#head(output$Phi, 3)
```

2.3 Model Parameters

Table 5 provides a general overview of all model parameters, the notations, and corresponding parameters in the ball example.

Table 5: General notation used to represent model parameters and an example of corresponding identifiers in `Phi` matrix.

Parameter	General Notation	Ball Example
True field calibration inputs	$\kappa = (\kappa_1, ..., \kappa_q)$	(`kappa1`)
Simulation GP scale	$\theta_s = (\theta_{s,1}, ..., \theta_{s,p+q})$	(`thetaS1 thetaS2`)
Simulation GP smoothness	$\alpha_s = (\alpha_{s,1}, ..., \alpha_{s,p+q})$	(`alphaS1 alphaS2`)
Bias-correction GP scale	$\theta_b = (\theta_{b,1}, ..., \theta_{b,p})$	(`thetaB1`)
Bias-correction GP smoothness	$\alpha_b = (\alpha_{b,1}, ..., \alpha_{b,p})$	(`alphaB1`)
Simulation marginal variance	σ_s^2	`sigma2S`
Bias-correction marginal variance	σ_b^2	`sigma2B`
Measurement error variance	σ_ϵ^2	`sigma2E`

Each row of matrix `Phi` represents a draw from joint distribution of parameters (MCMC run) and each column represents a parameter in the model. User-given initial values for parameters are used to initialize the first row of the `Phi` matrix. Then, each row will be used to find another sample from joint parameter space to fill the next row of `Phi` until matrix `Phi` is complete.

3. Parameter Estimation

`FBC` employs a full Bayesian approach to jointly estimate all parameters. To find the marginal posterior density distribution for each parameter, we need prior specification for each parameter (prior knowledge) and the joint likelihood estimation (full data).

3.1 Bayesian Analysis

Because `FBC` uses a full Bayesian framework, expert knowledge or opinion can be applied to the model parameters as prior specification. Variety of common prior distributions are implemented in `FBC` and can be used to specify the priors for each parameter (see section 1.2). There are seven classes of parameters and all have been specified in `calibrate()` using default values. Of those seven classes, calibration parameters (vector κ) and perhaps measurement error variance (scaler σ_ϵ^2) are application-dependent. It is recommended for user to specify the prior arguments for these parameters based on prior knowledge or consensus. Nevertheless, prior for calibration parameters is defaulted to Beta(1.1, 1.1) distribution [5]. It is close to standard uniform distribution (U(0, 1)) but densities approach to zero sharply as samples approach boundaries. This default

[5] *Note that the range of Beta distribution is the span of [0, 1], which is the range of calibration inputs for simulator after scaling .*

choice has been made to ensure a somewhat non-informative prior while de-emphasizing on boundary values[6]. For all other classes of parameters reasonable priors have been specified using default values. Priors for correlation scale parameters (vectors θ_s and θ_b) have been set to Gamma(1.1, 0.1) distribution. Similarly, the priors for correlation smoothness parameters (vectors α_s and α_b) have been set to Beta(5, 2) distribution that is shifted one unit to right to span [1, 2] as is the acceptable range for moothness parameters. This choice emphasizes higher (closer to 2 than 1) smoothness parameters. Finally, the priors for marginal simulator and bias-correction and measurement error variances (scalers σ_s^2, σ_b^2, and σ_ϵ^2) are set to be Inverse Gamma(1.5, 1.5). This emphasizes very low variances and de-emphasizes higher values.

By representing both data in a joint calibration model, we can compute the conditional likelihood of response given a parameter vector (See Appendix). Therefore, given the prior specifications above, we can derive joint posterior distribution of parameters given data:

$$\mathcal{P}[\phi|\mathcal{D}] \ \propto \ L(\mathcal{D}|\phi) \ . \ \mathcal{P}[\phi]$$

Where $\mathcal{D}$ represent full data (field and simulation). However, the above formulation is intractable and thus we need a simulation-based method to sample from joint posterior distribution. `FBC` implements a version of Markov Chain Monte Carlo (MCMC) simulation.

3.2 MCMC Simulation

MCMC simulation algorithm is used to draw samples from joint posterior distribution of the parameter space and build `Phi` matrix row by row. MCMC algorithm creates a Markov chain by updating parameters in each iteration according to a proposal scheme. Then given this parameter vector ($\phi^{(i)}$) and data ($\mathcal{D}$), the posterior likelihood can be computed. If posterior probability density of the parameters is larger than the density of a random draw from standard uniform distribution, the algorithm keeps the that parameter configuration by writing the next row of matrix `Phi`, otherwise updates the new row by last parameter vector. In either case, the new row will be used to generate the next proposal. Note that the first row of `Phi`, which is needed to start the algorithm, is supplied by user through initial values for the parameters. The detailed algorithm is presented in the Appendix.

3.3 Parameter Posterior Distributions

At the end of the MCMC run, sample of the joint posterior distribution for each parameter (a column in matrix `Phi`) can be used as an approximation of marginal posterior distribution. Center measures such as mean, mode, or median are provided as parameter estimate depending on the application and distribution shape. Furthermore, 50% and 80% credible sets are formed for each parameters to quantify the uncertainty in estimation. These statistics are provided in `summary` element in the output of `calibrate()` and additionally. (TODO: fix estimate->summary)

```
#output$estimates
```

Alternatively, posterior density kernels can be visualized over their assumed prior to investigate the effect of data on priors for each parameter.

```
# plot(output)
```

[6] *Boundary values of [0, 1] corresponds to $-\infty$ and ∞ in the original scale of calibration parameters.* .

4. Prediction

TODO

5. Implementation

TODO

6. Application

Ball Example

Data:

Spot Weld Example

Description:

Experimental Input: The physical model has three inputs: gauge (G), load (L), and current (C):

- Gauge (G):
- Load (L):
- Current (C):

Calibration Input: The simulation model has one additional input, τ that affects the amount of heat produced in the metal sheets. τ cannot be controlled in the physical experiment and its value is unknown. However it has to be specified for the simulation model as calibration input t:

- Heat generation factor τ: Factor affecting amount of heat produced

Mapping Parameters: To map the spot weld data (both field and simulation data) and parameters to `FBC` input configuration, we use the dagger † superscript to distinguish process parameters and variables with `FBC` variables and parameters:

$$\begin{aligned} \mathrm{x}_1 &\longrightarrow G^\dagger \\ \mathrm{x}_2 &\longrightarrow L^\dagger \\ \mathrm{x}_3 &\longrightarrow C^\dagger \\ \kappa_1 &\longrightarrow \tau^\dagger \end{aligned}$$

Kinetic Example

Appendix

MCMC Algorithm

Implementation of Metropolis within Gibbs algorithm

Let Φ be the matrix of parameter values (columns) indexed by MCMC iterations. Each column represents (after MCMC completes) the posterior density of a parameters. Since all parameters are included in Φ but have overlapping indices, the parameter densities (columns) are renamed to $(\phi_1, ..., \phi_d)$, where $d = 4p+3q+3$ is total number of parameters to have a unique index:

$$
\begin{array}{cccccccccc}
(\mathbf{1}) & & ... & & ... & & & & & (\mathbf{d}) \\
\mathbf{k}_1^* \ ... \ \mathbf{k}_q^* & & \mathbf{t}_{s1}^* \ ... \ \mathbf{t}_{s(p+q)}^* & & \mathbf{a}_{s1}^* \ ... \ \mathbf{a}_{s(p+q)}^* & & \mathbf{t}_{b1}^* \ ... \ \mathbf{t}_{bp}^* & & \mathbf{a}_{b1}^* \ ... \ \mathbf{a}_{bp}^* & \mathbf{v}_s^* \quad \mathbf{v}_b^* \quad \mathbf{v}_e^* \\
\Phi = \begin{bmatrix} k_1^{(1)} \ ... \ k_q^{(1)} \end{bmatrix} & & \begin{bmatrix} t_{s1}^{(1)} \ ... \ t_{s(p+q)}^{(1)} \end{bmatrix} & & \begin{bmatrix} a_{s1}^{(1)} \ ... \ a_{s(p+q)}^{(1)} \end{bmatrix} & & \begin{bmatrix} t_{b1}^{(1)} \ ... \ t_{bp}^{(1)} \end{bmatrix} & & \begin{bmatrix} a_{b1}^{(1)} \ ... \ a_{bp}^{(1)} \end{bmatrix} & \begin{bmatrix} v_s^{(1)} \quad v_b^{(1)} \quad v_e^{(1)} \end{bmatrix} \\
\begin{bmatrix} ... \end{bmatrix} & & \begin{bmatrix} ... \end{bmatrix} & & \begin{bmatrix} ... \end{bmatrix} & & \begin{bmatrix} ... \end{bmatrix} & & \begin{bmatrix} ... \end{bmatrix} & \begin{bmatrix} ... \end{bmatrix} \\
\begin{bmatrix} k_1^{(N)} \ ... \ k_q^{(N)} \end{bmatrix} & & \begin{bmatrix} t_{s1}^{(N)} \ ... \ t_{s(p+q)}^{(N)} \end{bmatrix} & & \begin{bmatrix} a_{s1}^{(N)} \ ... \ a_{s(p+q)}^{(N)} \end{bmatrix} & & \begin{bmatrix} t_{b1}^{(N)} \ ... \ t_{b(p+q)}^{(N)} \end{bmatrix} & & \begin{bmatrix} a_{b1}^{(N)} \ ... \ a_{b(p+q)}^{(N)} \end{bmatrix} & \begin{bmatrix} v_s^{(N)} \quad v_b^{(N)} \quad v_e^{(N)} \end{bmatrix}
\end{array}
$$

- Initialize the first row with user-given initial values:

$$
\begin{aligned}
\phi^{(1)} &= (\phi_1^{(1)},\ ...,\ \phi_d^{(1)}) \\
&= (k_1,\ ...,\ k_q,\ \theta_0^{(1)},\ ...,\ \theta_0^{(p+q)},\ \alpha_0^{(1)},\ ...,\ \alpha_0^{(p+q)},\ \theta_0^{(1)},\ ...,\ \theta_0^{(p)},\ \alpha_0^{(p)},\ ...,\ \alpha_0^{(p)},\ v_{s0},\ v_{b0},\ v_{e0})
\end{aligned}
$$

- At each iteration $i \in (2, ..., N)$, and to update j-th parameter ($j \in (1, ..., d)$ and first $(j-1)$ parameters are already updated):

  1. Propose a new value for $\phi_j^{(i)}$ based on its last update [7]. $\phi_j^{(i-1)}$, called Metropolis Update (MU):

$$\phi_j^* = \mathcal{N}(\phi_j^{(i-1)}, \sigma_p^2)$$

     where σ_p^2 is adaptively adjusted [8] to ensure (faster) convergence.

  2. Form the parameter vector ϕ^* based on MU:

$$
\begin{aligned}
\phi^{(last)} &= (\phi_1^{(i)}, ..., \phi_{j-2}^{(i)},\ \phi_{j-1}^{(i)},\ \phi_j^{(i-1)},\ ...,\ \phi_d^{(i-1)}) \\
\phi^* \quad &= (\phi_1^{(i)}, ..., \phi_{j-1}^{(i)},\ \phi_j^*,\ \phi_{j+1}^{(i-1)},\ ...,\ \phi_d^{(i-1)})
\end{aligned}
$$

  3. Draw a random sample u from $U(0, 1)$ and take its log: $\ln(u)$

[7] If it is first parameter, the last update is the last row ($\phi^{(i-1)}$

[8] Every 50 iterations, acceptance rate (AR) is computed. If $AR < 0.44$, proposal variance σ_p^2 is decreased, and vice versa.It has been shown that for one-dimensional proposals used in Metropolis within Gibbs algorithm, the optimal acceptance rate is 0.44 (TODO: citation).

4. Compute the difference between the log of joint posterior density given current and last parameter vectors:

$$h(\phi^*,\ \phi^{(last)}) = \ln(L(\mathrm{y}|\phi^*)) + \ln(\mathcal{P}[\phi^*]) - \ln(L(\mathrm{y}|\phi^{(last)})) + \ln(\mathcal{P}[\phi^{(last)}])$$

5. If $h(\phi^*,\ \phi^{(last)}) > \ln(u)$, set:

$$\phi_j^{(i)} = \phi_j^*$$

otherwise,

$$\phi_j^{(i)} = \phi_j^{(i-1)}$$

6. The update vector now is:

$$\phi^{(last)} = (\phi_1^{(i)}, ..., \phi_{j-1}^{(i)},\ \phi_j^{(i)},\ \phi_{j+1}^{(i-1)},\ ...,\ \phi_d^{(i-1)})$$

- If all parameters are updated, go to next iteration of i
- When iterations of i is completed, return the matrix of Φ that contains joint posterior density distribution of all parameters. Marginal distribution of each parameter can be used for point prediction and uncertainty quantification (credible interval).

Bayesian Analysis

In the Bayesian framework, the joint probability distribution of all parameters and hyperparameters of the calibration model given data ($\mathcal{P}[\phi|\mathrm{y}]$) can be derived:

$$L(\mathrm{y}|\phi) = |C|^{-\frac{1}{2}} \,.\, e^{-\frac{1}{2}\mathrm{y}.C^{-1}.\mathrm{y}^T}$$

$$\mathcal{P}[\phi|\mathrm{y}] \ \propto\ L(\mathrm{y}|\phi) \,.\, \mathcal{P}[\phi]$$

Taking the log from both sides will decrease computational load and increase speed:

$$\ln(L(\mathrm{y}|\phi)) = -\frac{1}{2}\ln(|C|) - \frac{1}{2}\mathrm{y}.C^{-1}.\mathrm{y}^T$$

$$\begin{aligned}
\ln(\mathcal{P}[\phi|\mathrm{y}]) \ &\propto \ln(L(\mathrm{y}|\phi)) + \ln(\mathcal{P}[\phi) \\
&= -\frac{1}{2}\ln(|C|) - \frac{1}{2}\mathrm{y}.C^{-1}.\mathrm{y}^T && \text{(log liklihood given full data: X, y)} \\
&+ \sum_{i=1}^{q} \mathcal{P}[\kappa_i] && \text{(priors for calibration parameters)} \\
&+ \sum_{i=1}^{p+q} \mathcal{P}[\theta_{si}] + \sum_{i=1}^{p+q} \mathcal{P}[\alpha_{si}] \ + \ \mathcal{P}[\sigma_s^2] && \text{(priors for } \eta(.) \text{ hyperparameters)} \\
&+ \sum_{i=1}^{p} \mathcal{P}[\theta_{bi}] + \sum_{i=1}^{p} \mathcal{P}[\alpha_{bi}] \ + \ \mathcal{P}[\sigma_b^2] && \text{(priors for } \delta_\kappa(.) \text{ hyperparameters)} \\
&+ \ \mathcal{P}[\sigma_\epsilon^2] && \text{(prior for measurement error variance)}
\end{aligned}$$

Above equation is intractable and thus a simulation-based method must be used to sample from posterior distribution. `FBC` implements a version Markov Chain Monte Carlo (MCMC) simulation.

Full Model

KOH model combines simulation and field data to form a joint model using joint dataset:

$$\mathbf{y} \ = \begin{bmatrix} \mathbf{y_f} \\ \mathbf{y_s} \end{bmatrix} = (y_1, \ \ldots \ , \ y_{n+m})^T \qquad \text{(joint vector of responses)}$$

$$\mathbf{X} = \begin{bmatrix} \mathbf{X}_\kappa \\ \mathbf{X_s} \end{bmatrix} = \begin{bmatrix} \mathbf{x_1} \ \mathbf{x_2} \ \ldots \ \mathbf{x_{p+q}} \end{bmatrix} \qquad \text{(joint input matrix)}$$

$$\mathbf{x_i} = (x_1, x_2, ..., x_{n+m}) \quad \forall i \in \{1, 2, ..., p+q\}$$

Since $\mathbf{X_f}$ is a sub-matrix of $\mathbf{X}$, we can represent the functional relationship between input and response for full model with $\zeta(.)$, which is considered to be realization of a random function and derived from $\eta(.)$ and $\delta_\kappa(.)$:

$$\mathbf{y} \ = \zeta(\mathbf{X})$$

Because both $\eta(.)$ and $\delta_\kappa(.)$ are GPs, $\zeta(.)$ can also be considered a zero mean GP:

$$\zeta(.) \sim GP \ (0, \ C(.,.))$$

Where covariance matrix C is dependent to full input matrix $\mathbf{X}$ and hyperparameters of $\eta(.)$ and $\delta_\kappa(.)$, which in turn are dependent to model hyperparameters.

Using matrix notation, input/output relationship of both experiments is presented below in matrix notation. This relations can be used to derive a relationship for joint data:

$$\mathrm{y} \ = \zeta(X) = \eta(X) + \begin{bmatrix} \delta_\kappa(X_f) + \mathcal{E} & 0 \\ 0 & 0 \end{bmatrix}$$

Since X_f is a subset of matrix X, the joint response y can be modeled as a random function $\zeta(X)$. In the ball example, the full input matrix X is a (163×2) matrix by stacking X_κ on X_s Note that since κ is unknown, the initial value `c0` is used internally to build X_κ.

Where, C_η and C_δ are covariance matrices of full input matrix X and original field input matrix X_f. And C is characterized as:

$$C(X, X) = C_\eta(X, X) + \begin{bmatrix} C_\delta(X_f, X_f) + \sigma_\epsilon^2.I_n & 0 \\ 0 & 0 \end{bmatrix}$$

$$= \begin{bmatrix} C_\eta(X_f, X_f) + C_\delta + \sigma_\epsilon^2.I_n & C_\eta(X_s, X_f) \\ C_\eta(X_f, X_s) & C_\eta(X_s, X_s) \end{bmatrix}$$

Note that C_s covariance matrix of the full data X is further divided to components $C_\eta(X_f, X_f)$, $C_\eta(X_s, X_f)$, $C_\eta(X_f, X_s)$, and $C_\eta(X_s, X_s)$ to optimize the computation.

References