Creating a simple emulator case study from scratch: a cookbook

Robin K. S. Hankin
University of Cambridge

Abstract

This document constructs a minimal working example of a simple application of the **emulator** package, step by step. Datasets and functions have a `.vig` suffix, representing "vignette".

Keywords: emulator, BACCO, R.

1. Introduction

Package **emulator** of bundle **BACCO** performs Bayesian emulation of computer models. This document constructs a minimal working example of a simple problem, step by step. Datasets and functions have a `.vig` suffix, representing "vignette".

This document is not a substitute for Kennedy and O'Hagan (2001a) or Kennedy and O'Hagan (2001b) or Hankin (2005) or the online help files in **BACCO**. It is not intended to stand alone: for example, the notation used here is that of Kennedy and O'Hagan (2001a,b), and the user is expected to consult the online help in the **BACCO** package when appropriate.

This document is primarily didactic, although it is informal.

Nevertheless, many of the points raised here are duplicated in the **BACCO** helpfiles.

The author would be delighted to know of any improvements or suggestions. Email me at `hankin.robin@gmail.com`.

2. List of objects that the user needs to supply

The user needs to supply three objects:

- A design matrix, here `val.vig` (rows of this show where the code has been evaluated)
- Basis functions. Here `basis.vig()`. This shows the basis functions used for fitting the prior
- Data, here `z.vig`. This shows the data obtained from evaluating the various levels of code at the points given by the design matrix and the subsets object.

Each of these is discussed in a separate subsection below.

But the first thing we need to do is install the library:

2.1. Design matrix: USER TO SUPPLY

In these sections I show the objects that the user needs to supply, under a heading like the one above. In the case of the `emulator` we need a design matrix and a vector of outputs.

The first thing needed is the design matrix `val.vig`, ie the points in parameter space at which the lowest-level code is executed. The example here has just two parameters, `a` and `b`:

```
> head(val.vig)
```

```
           [,1]       [,2]
[1,] 0.01666667 0.08333333
[2,] 0.81666667 0.88333333
[3,] 0.41666667 0.85000000
[4,] 0.78333333 0.41666667
[5,] 0.48333333 0.28333333
[6,] 0.31666667 0.71666667
```

```
> nrow(val.vig)
```

```
[1] 30
```

Notes

- Each row is a point in parameter space, here two dimensional.
- The parameters are labelled `a` and `b`

2.2. Basis functions: USER TO SUPPLY

Now we need to choose a basis function. Do this by copying `basis.toy()` but fiddling with it:

```
> basis.vig <- function(x) {
+     out <- c(1, x, x[1] * x[2])
+     names(out) <- c("const", LETTERS[1:2], "interaction")
+     return(out)
+ }
```

Notes

- This is shamelessly ripped off from `basis.toy()`, except that I've changed the basis to be `c(1,a,b,ab)`.
- in the function, `out` is a vector of length four: `c(1,x[1],x[2], x[1]*x[2])`.

2.3. Data: USER TO SUPPLY

The data we have for the `.vig` example is a vector whose elements are the output of the code at the points specified in `val.vig`:

```
> head(z.vig)

[1] 0.9534657 7.9293059 5.9899320 4.5833130 2.9650400 5.2386355

> summary(z.vig)

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
 0.9535  2.8450  4.5450  4.4580  5.6260  8.4300 
```

3. Data analysis

The previous section showed what data and functions the user needs to supply. These all have a `.vig` suffix. This section shows the data being analyzed.

First we will estimate the scales to use:

```
> os <- optimal.scales(val = val.vig, scales.start = c(10, 10),
+     d = z.vig, func = basis.vig)
> os

[1] 2.188349 4.623837
```

So we can estimate the coefficients. But first we have to calculate the variance matrix and invert it:

```
> A.os <- corr.matrix(xold = val.vig, scales = REAL.SCALES)
> Ainv.os <- solve(A)
```

Given this, use `betahat.fun()` to get the coeffs:

```
> betahat.fun(xold = val.vig, d = z.vig, Ainv = solve(A), func = basis.vig)

      const           A           B interaction 
  1.8534363   0.9513379   2.4648857   4.6930890 
```

The central function is interpolant:

```
> interpolant(x = c(0.5, 0.5), d = z.vig, Ainv = Ainv.os, scales = os,
+     xold = val.vig, func = basis.vig, give.full.list = TRUE)

$betahat
      const           A           B interaction 
  1.8534363   0.9513379   2.4648857   4.6930890 

$prior
```

```
        [,1]
[1,] 4.73482

$beta.var
                  const          A          B interaction
const         0.2326390 -0.1975772 -0.2431482   0.2202426
A            -0.1975772  0.4706564  0.2172804  -0.5217411
B            -0.2431482  0.2172804  0.5066074  -0.4550997
interaction   0.2202426 -0.5217411 -0.4550997   0.9821526

$beta.marginal.sd
      const           A           B interaction
  0.4823267   0.6860440   0.7117636   0.9910361

$sigmahat.square
[1] 0.3537308

$mstar.star
         [,1]
[1,] 4.347081

$cstar
[1] -0.0671719

$cstar.star
[1] -0.04480336

$Z
[1] 0.1258901
```

And that's it, really.

References

Hankin RKS (2005). "Introducing **BACCO**, an R bundle for Bayesian analysis of computer code output." *Journal of Statistical Software*, **14**(16).

Kennedy MC, O'Hagan A (2001a). "Bayesian calibration of computer models." *Journal of the Royal Statistical Society, Series B*, **63**(3), 425–464.

Kennedy MC, O'Hagan A (2001b). "Supplementary details on Bayesian calibration of computer models." Internal Report. URL http://www.shef.ac.uk/~st1ao/ps/calsup.ps.

A. Data generation

The data used in this study were created by directly sampling from the appropriate multivariate Gaussian:

```
> REAL.BETA <- 1:4
> REAL.SCALES <- c(3, 6)
> REAL.SIGMASQUARED <- 0.3
> A <- corr.matrix(xold = val.vig, scales = REAL.SCALES)
> z.vig <- as.vector(rmvnorm(n = 1, mean = crossprod(REAL.BETA,
+     apply(val.vig, 1, basis.vig)), sigma = A * REAL.SIGMASQUARED))
```

Affiliation:

Robin K. S. Hankin
University of Cambridge
19 Silver Street Cambridge CB3 9EP United Kingdom
E-mail: hankin.robin@gmail.com