

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,] 0.6166667 0.6166667
[2,] 0.2166667 0.3833333
[3,] 0.5166667 0.9833333
[4,] 0.9833333 0.1833333
[5,] 0.1500000 0.8166667
[6,] 0.4833333 0.5500000

> 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] 6.127404 2.891223 6.898828 4.467377 4.026445 5.099659
```

```
> summary(z.vig)
```

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
1.409	2.892	4.604	4.742	6.333	8.909

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] 3.912712 5.564970
```

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
1.4873454	0.7715458	2.0767787	5.7623741

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
1.4873454	0.7715458	2.0767787	5.7623741

```
$prior
```

```

      [,1]
[1,] 4.352101

$beta.var
      const      A      B interaction
const  0.3259895 -0.3123325 -0.3525850  0.3653226
A      -0.3123325  0.6003079  0.3750083 -0.7197591
B      -0.3525850  0.3750083  0.6616136 -0.7045869
interaction 0.3653226 -0.7197591 -0.7045869  1.3947705

$beta.marginal.sd
      const      A      B interaction
0.5709549  0.7747954  0.8133963  1.1810040

$sigma.hat.square
[1] 0.3781388

$mstar.star
      [,1]
[1,] 4.842632

$cstar
[1] -0.0514832

$cstar.star
[1] -0.05021579

$Z
[1] 0.1377989

```

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