Creating a simple approximator 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 **approximator** package, step by step. Datasets and functions have a `.vig` suffix, representing "vignette".

Keywords: **emulator**, **approximator**, **BACCO**, R.

1. Introduction

Package **approximator** of bundle **BACCO** performs Bayesian calibration of computer models when fast approximations are available. 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 (2000) or Hankin (2005) or the online help files in **approximator**. It is not intended to stand alone: for example, the notation used here is that of Kennedy and O'Hagan (2000), and the user is expected to consult the online help in the **approximator** 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.

Note that many of the objects created in this document are interdependent and changing one sometimes implies changing many others.

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 five objects:

- A design matrix, here `D1.vig` (rows of this show where code level 1 has been evaluated)
- A subset object, in the form of a list. Here it is `subsets.vig`. This list has one element per level of code. A subset object shows which points in the design matrix have been evaluated at each level.

- 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.
- A hyperparameter object, here `hpa.vig`. This object holds correlation scales, the rhos, and the sigmas. One convenient way to do this is to define a function that creates a hyperparameter object from a vector; an example is given in the appendix (`hpa.fun.vig()`) but this is not strictly necessary.

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 `approximator` package, the objects have a simple structure (list of vectors, function, etc) and so I just show what they look like.

The first thing needed is the design matrix `D1.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(D1.vig)

        [,1]    [,2]
[1,] 0.90625 0.46875
[2,] 0.21875 0.65625
[3,] 0.34375 0.84375
[4,] 0.46875 0.28125
[5,] 0.65625 0.59375
[6,] 0.15625 0.90625

> nrow(D1.vig)

[1] 16
```

Notes

- Each row is a point in parameter space, here two dimensional. The bottom level code is run at each of these points (see `subsets.vig`)
- The parameters are labelled `a` and `b`

2.2. Subsets object: USER TO SUPPLY

We now need a `subsets.object`, which gives the row numbers of the runs at each level.

```
> subsets.vig
```

```
$level.1
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16

$level.2
[1]  1  2  3  7  9 13 14 16

$level.3
[1]  1  7 13 14 16

$level.4
[1]  1  7 13 16
```

Notes

- This is a list of 4 elements (ie the number of levels)
- each element is a subset of those above it
- Use functions `is.nested()` and `is.strict()` to verify that the subsets are consistent.

2.3. 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) {
+     if (is.vector(x)) {
+         stopifnot(length(x) == 2)
+         out <- c(1, x, x[1] * x[2])
+         names(out) <- c("const", LETTERS[1:2], "interaction")
+         return(out)
+     }
+     else {
+         return(t(apply(x, 1, match.fun(sys.call()[[1]]))))
+     }
+ }
```

Notes

- This is shamelessly ripped off from `basis.toy()`, except that I've changed the basis to be `c(1,a,b,ab)`.
- Also note the rather strange way this function deals with vectors and matrices. Vectors via the first bit, and matrices via that strange `apply()` bit at the end.
- The line that reads `stopifnot(length(x) == 2)` is there to ensure that only vectors of length 2, or matrices with two columns, are processed.

Here is an example of `basis.vig()` in action:

```
> head(basis.vig(D1.vig))

     const       A       B interaction
[1,]     1 0.90625 0.46875   0.4248047
[2,]     1 0.21875 0.65625   0.1435547
[3,]     1 0.34375 0.84375   0.2900391
[4,]     1 0.46875 0.28125   0.1318359
[5,]     1 0.65625 0.59375   0.3896484
[6,]     1 0.15625 0.90625   0.1416016
```

See how the columns become the basis functions (that is, `c(1, x , x[1]*x[2])`).

2.4. Data: USER TO SUPPLY

The data needed is output from the four levels of code. The code is evaluated at points specified by the design matrix `D1.vig`.

Code level 1 is evaluated at each point of `D1.vig` (ie `D1.vig[subsets[[1]],]`)

Code level 2 is evaluated at `D1.vig[subsets[[2]],]`

Code level `n` is evaluated at `D1.vig[subsets[[n]],]`

The data we have for the `.vig` example is a list of four elements. Each element is a vector whose ith element is the code output at the appropriate point in the design matrix. We can get a feel for the dataset by looking at the head of each vector, and extracting the length:

```
> lapply(z.vig, head)

[[1]]
[1] 5.915625 3.965019 5.244055 3.370008 5.582936 4.571371

[[2]]
[1] 10.905791  7.822174 10.335172  5.331106  5.736572  9.215646

[[3]]
[1] 14.963542  7.343231 12.702240 14.192522  7.657568

[[4]]
[1] 17.76041  9.15108 15.21075  9.32725

> lapply(z.vig, length)

[[1]]
[1] 16

[[2]]
```

```
[1] 8

[[3]]
[1] 5

[[4]]
[1] 4
```

2.5. Hyperparameters: USER TO SUPPLY

We now need some hyperparameters. The appendix gives an example of how to specify a function that creates a hyperparameter object. Here I will show an example

```
> hpa.vig

$sigma_squareds
level1 level2 level3 level4
  0.01   0.01   0.01   0.01

$B
$B[[1]]
   A  B
A 20  0
B  0 20

$B[[2]]
   A  B
A 20  0
B  0 20

$B[[3]]
   A  B
A 20  0
B  0 20

$B[[4]]
   A  B
A 20  0
B  0 20


$rhos
level1 level2 level3
     1      1      1
```

Notes

- The hyperparameter object is a list of three elements:
  - The first element, `sigma_squareds`, is a vector of variances (one per level)
  - The second element is a list of length n (the number of levels). Each of these elements is a positive definite matrix of correlation lengths (here diagonal for simplicity)
  - The third element is a vector of length $n - 1$ of the rhos.
- Different problems will have different hyperparameter objects
- In this case it's probably easier to create a hyperparameter object by hand, but in the appendix I show how a function to generate hyperparameter objects may be written. This option is sometimes better.

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.

3.1. Estimate of the coefficients in the hyperparameter object

This estimate uses the initial value for the hyperparameters.

The hyperparameters themselves may be estimated by using functions `opt.1()` and `opt.gt.1()` for level 1 and levels 2 and greater, respectively.

```
> jj <- list(trace = 100, maxit = 10)
> hpa.vig.level1 <- opt.1(D = D1.vig, z = z.vig, basis = basis.vig,
+     subsets = subsets.vig, hpa.start = hpa.vig, control = jj)

  Nelder-Mead direct search function minimizer
function value for initial parameters = -76.604986
  Scaled convergence tolerance is 1.1415e-06
Stepsize computed as 0.460517
BUILD              3 -71.320088 -76.604986
EXTENSION          5 -74.189195 -81.766288
REFLECTION         7 -76.604986 -82.991534
LO-REDUCTION       9 -81.766288 -83.249197
Exiting from Nelder Mead minimizer
    11 function evaluations used

> hpa.vig.level1

$sigma_squareds
     level1      level2      level3      level4
0.002511886 0.010000000 0.010000000 0.010000000
```

```
$B
$B[[1]]
         A        B
A 39.90525  0.00000
B  0.00000 39.90525

$B[[2]]
   A  B
A 20  0
B  0 20

$B[[3]]
   A  B
A 20  0
B  0 20

$B[[4]]
   A  B
A 20  0
B  0 20


$rhos
level1 level2 level3
     1      1      1
```

Notes

- Function `opt.1()` takes a whole bunch o' inputs and returns a modified hyperparameter object.
- The hyperparamer object that `opt.1()` returns is identical to `hpa.start` except for `sigma_squareds[1]` and `B[[1]]`, corresponding to the first level.
- We can use this output as a start point to functions `opt.gt.1()` et seq

```
> jj <- list(trace = 0, maxit = 4)
> hpa.vig.level2 <- opt.gt.1(level = 2, D = D1.vig, z = z.vig,
+     basis = basis.vig, subsets = subsets.vig, hpa.start = hpa.vig.level1,
+     control = jj)
> hpa.vig.level3 <- opt.gt.1(level = 3, D = D1.vig, z = z.vig,
+     basis = basis.vig, subsets = subsets.vig, hpa.start = hpa.vig.level2,
+     control = jj)
> hpa.vig.level4 <- opt.gt.1(level = 4, D = D1.vig, z = z.vig,
+     basis = basis.vig, subsets = subsets.vig, hpa.start = hpa.vig.level3,
+     control = jj)
> hpa.vig.level4
```

```
$sigma_squareds
     level1      level2      level3      level4
0.002511886 0.010000000 0.010000000 0.010000000

$B
$B[[1]]
         A        B
A 39.90525  0.00000
B  0.00000 39.90525

$B[[2]]
         A        B
A 31.69786  0.00000
B  0.00000 31.69786

$B[[3]]
         A        B
A 31.69786  0.00000
B  0.00000 31.69786

$B[[4]]
   A  B
A 20  0
B  0 20


$rhos
level1 level2 level3
     1      1      1
```

Now we can try and estimate the betas using the optimized hyperparameter object:

```
> betahat.app(D1 = D1.vig, subsets = subsets.vig, basis = basis.vig,
+     hpa = hpa.vig.level4, z = z.vig)

      level1.const          level1.A          level1.B level1.interaction
        1.10666125         1.90301085         2.83912840         4.11716461
      level2.const          level2.A          level2.B level2.interaction
        1.33837779         0.53484911         2.66313188         4.60394349
      level3.const          level3.A          level3.B level3.interaction
        1.61544206        -0.06345057         0.24695743         5.52510323
      level4.const          level4.A          level4.B level4.interaction
        0.79059196         1.59233103         0.94773623         0.28006421
```

Not too bad.

3.2. The package in use

The final stage would be using function `mdash.fun()`, which gives the posterior expectation of the Gaussian process (for level 4):

```
> mdash.fun(x = c(0.5, 0.5), D1 = D1.vig, subsets = subsets.vig,
+     hpa = hpa.vig.level4, z = z.vig, basis = basis.vig)

[1] 13.8142
```

We can now give an error estimate here:

```
> cdash.fun(x = c(0.5, 0.5), D1 = D1.vig, subsets = subsets.vig,
+     basis = basis.vig, hpa = hpa.vig)

          [,1]
[1,] 0.1093078
```

Appendix

A. Data generation

In practice the user generates data from a climate model. Here, I will generate data that matches the assumptions of the approximator software exactly.

Now we need a design matrix:

```
> n <- 16
> set.seed(0)
> D1.vig <- latin.hypercube(n, 2)
```

See how the function `latin.hypercube()` is used.

Now we need to specify which rows of the design matrix are run at each of the various levels. We can generate some of this randomly. The lowest level code will use all rows of `D1.vig`, level 2 will use about half of them, level 3 (ie the top level) will use about half of them, and level 4 about half of *them:*

```
> subsets.vig <- subsets.fun(n, levels = 4, prob = 0.6)
> names(subsets.vig) <- paste("level", 1:4, sep = ".")
```

Notes

- See the `.vig` suffix, for "vignette".
- randomly chosen values illustrate the general nature of the software

Notes

- The `is.vector()` test allows one to treat matrices and vectors in a consistent way
- the last line is a kludge, but no better way seems to exist

Now we need a function that creates a hyperparameter object:

```
> hpa.fun.vig

function (x)
{
    if (length(x) != 15) {
        stop("x must have 19 elements")
    }
    "pdm.maker" <- function(x) {
        jj <- diag(x[1:2], nrow = 2)
        rownames(jj) <- LETTERS[1:2]
        colnames(jj) <- LETTERS[1:2]
```

```
        return(jj)
    }
    sigma_squareds <- x[1:4]
    names(sigma_squareds) <- paste("level", 1:4, sep = "")
    B <- list()
    B[[1]] <- pdm.maker(x[5:6])
    B[[2]] <- pdm.maker(x[7:8])
    B[[3]] <- pdm.maker(x[9:10])
    B[[4]] <- pdm.maker(x[11:12])
    rhos <- x[13:15]
    names(rhos) <- paste("level", 1:3, sep = "")
    return(list(sigma_squareds = sigma_squareds, B = B, rhos = rhos))
}
```

Notes:

- This is a modification of `hpa.fun.toy()` but with a smaller number of params
- This function isn't strictly necessary, but the alternative (defining a hyperparameter object *ab initio*) is fiddly and error-prone.

Now we can call this function and create a hyperparameter object

```
> hpa.vig <- hpa.fun.vig(c(rep(0.01, 4), rep(20, 8), rep(1, 3)))
```

Now we can generate some data. In practice, this data will come from a climate model. Here I will cheat and define a function `generate.vig.observations()` to generate data that comes from a known distribution:

First define a function ripped off from `generate.toy.obs()`:

```
> "generate.vig.observations" <- function(D1, subsets, basis.fun,
+     hpa, betas = NULL, export.truth = FALSE) {
+     if (is.null(betas)) {
+         betas <- rbind(c(1, 2, 3, 4), c(1, 1, 3, 4), c(1, 1,
+             1, 4), c(1, 1, 1, 1))
+         colnames(betas) <- c("const", LETTERS[1:2], "interaction")
+         rownames(betas) <- paste("level", 1:4, sep = "")
+     }
+     if (export.truth) {
+         return(list(hpa = hpa, betas = betas))
+     }
+     sigma_squareds <- hpa$sigma_squareds
+     B <- hpa$B
+     rhos <- hpa$rhos
+     delta <- function(i) {
+         out <- rmvnorm(n = 1, mean = basis.fun(D1[subsets[[i]],
+             , drop = FALSE]) %*% betas[i, ], sigma = sigma_squareds[i] *
```

```
+             corr.matrix(xold = D1[subsets[[i]], , drop = FALSE],
+                 pos.def.matrix = B[[i]]))
+         out <- drop(out)
+         names(out) <- rownames(D1[subsets[[i]], , drop = FALSE])
+         return(out)
+     }
+     use.clever.but.untested.method <- FALSE
+     if (use.clever.but.untested.method) {
+         z1 <- delta(1)
+         z2 <- delta(2) + rhos[1] * z1[match(subsets[[2]], subsets[[1]])]
+         z3 <- delta(3) + rhos[2] * z2[match(subsets[[3]], subsets[[2]])]
+         z4 <- delta(4) + rhos[3] * z3[match(subsets[[4]], subsets[[3]])]
+         return(list(z1 = z1, z2 = z2, z3 = z3, z4 = z4))
+     }
+     else {
+         out <- NULL
+         out[[1]] <- delta(1)
+         for (i in 2:length(subsets)) {
+             out[[i]] <- delta(i) + rhos[i - 1] * out[[i - 1]][match(subsets[[i]],
+                 subsets[[i - 1]])]
+         }
+         return(out)
+     }
+ }
```

Then call it:

```
> z.vig <- generate.vig.observations(D1 = D1.vig, subsets = subsets.vig,
+     basis.fun = basis.vig, hpa = hpa.vig)

> z.vig

[[1]]
 [1] 5.915625 3.965019 5.244055 3.370008 5.582936 4.571371 2.985063 4.081773
 [9] 3.274803 5.487250 3.370762 1.351710 4.998310 5.481208 8.344171 2.910165

[[2]]
[1] 10.905791  7.822174 10.335172  5.331106  5.736572  9.215646 10.413659
[8]  5.491027

[[3]]
[1] 14.963542  7.343231 12.702240 14.192522  7.657568

[[4]]
[1] 17.76041  9.15108 15.21075  9.32725
```

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 (2000). "Predicting the output from a complex computer code when fast approximations are available." *Biometrika*, **87**(1), 1–13.

Affiliation:

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