Creating a simple `calibrator` case study from scratch: a cookbook

Robin K. S. Hankin

March 7, 2007

1 Introduction

Package `calibrator` of bundle `BACCO` performs Bayesian calibration of computer models. This document constructs a minimal working example of a simple problem, step by step. Datasets and functions have a `.int` suffix, representing "intermediate".

This document is not a substitute for KOH or KOHa 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 KOH, 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.

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 `r.hankin@noc.soton.ac.uk`.

Observations are made over two parameters, x and y, which I style "latitude" and "longitude". I tend to think of the observation as temperature.

The model requires two parameters, A and B and, given these, gives output as a function of x and y. The model is thus a function of four variables: x, y, A, B. Write $\theta = (A, B)$ to denote the two parameters collectively. KOH use $\mathbf{t}$ to denote a specific value of parameters, and θ to denote any old set of parameters one might like to consider.

Parameters A and B have true but unknown values and we wish to make inferences about these true values.

2 List of objects that the user needs to supply

The user needs to supply several objects:

- Design matrices for the code runs and the field observations, here `D1.int` and `D2.int`.
- An extractor function to separate out the parameters from the independent variables, here `extractor.int()`
- Various basis functions for the code runs (here `h1.int()` and `H1.int()`) and field observations (here `h2.int()` and `H2.int()`)
- A function to create a hyperparameter object, here `phi.fun.int()`
- Two functions to return expectation under different distributions, here `E.theta.int()` and `Edash.theta.int()`
- Data, here `y.int` for the code observations and `z.int` for the field observations, and `d.int` for both together

Objects (data and functions) in the package follow a consistent nomenclature: anything that ends in "`.toy`" is specific to the toy datasets, and a replacement needs to be supplied by the user (in this vignette, the ending "`.int`" is used). Anything that does not end with "`.toy`" is generic, in the sense that it will work for any application and does not need be supplied by the user, and should not be altered. For example, `hh.fun()` is generic because it does not end with `.toy`.

3 Design matrices

There are two design matrices to consider: `D1.int`, which is the code run set, and `D2.int`, the observation points. Generating the data is deferred to section 6 because we need basis functions (section 3) and hyperparameters (section 4) before we can specify the appropriate distribution from which random can be drawn.

In this section, we will generate design matrices of arbitrary size. Data is generated in section 7.

First, define the number of code observations (`n1`) and the number of field observations (`n2`):

```
> library(calibrator)
> library(emulator)

> n1 <- 20
> n2 <- 21
```

These can be varied at will. Now create the D1 matrix, of code observation points. This will consist of

```
> D1.int <- latin.hypercube(n1, 4)
> rownames(D1.int) <- paste("coderun", 1:nrow(D1.int), sep = ".")
> colnames(D1.int) <- c("x", "y", "A", "B")
> head(D1.int)

               x     y     A     B
coderun.1 0.075 0.925 0.825 0.975
coderun.2 0.025 0.175 0.225 0.575
coderun.3 0.775 0.075 0.475 0.325
coderun.4 0.425 0.675 0.175 0.625
coderun.5 0.675 0.325 0.325 0.475
coderun.6 0.475 0.425 0.075 0.175
```

Notes

- Rownames and column-names are applied. They are not strictly necessary, but are very useful in debugging.
- the points are randomly chosen (but fill parameter space reasonably)
- We have ten observation points. More can be added but at the cost of slower run times.

Now the field observation points. This is a two-column matrix, with an x column and a y column.

```
> D2.int <- latin.hypercube(n2, 2)
> rownames(D2.int) <- paste("obs", 1:nrow(D2.int), sep = ".")
> colnames(D2.int) <- c("x", "y")
> head(D2.int)
```

```
               x          y
obs.1 0.64285714 0.54761905
obs.2 0.88095238 0.35714286
obs.3 0.59523810 0.21428571
obs.4 0.97619048 0.40476190
obs.5 0.50000000 0.16666667
obs.6 0.07142857 0.02380952
```

Notes

- See how the number of observations (n1) is different from the number of code runs (n2) as an additional safety check.
- Rownames and columnnames are given.

3.1 Extractor functions

Now we need a function to extract the variable part and the parameter part. Working by analogy to `extractor.toy()`:

```
> extractor.int <- function(D1) {
+     return(list(x.star = D1[, 1:2, drop = FALSE], t.vec = D1[,
+         3:4, drop = FALSE]))
+ }
```

Notes

- The function returns a two-element list, named `x.star` and `t.vec`.
- It just extracts the relevant columns: the first two give the lat and long, and columns three and four give A and B
- `drop=FALSE` is needed to deal with one-row dataframes consistently.

4 Basis functions

We now need basis functions. The basis function `h1.toy()` is just a constant term with each component. We will use a more sophisticated version.

Recall the fundamental equation of KOH:

$$z(\mathbf{x}) = \rho\eta(\mathbf{x}, \theta) + \delta(\mathbf{x}) + \epsilon \tag{1}$$

where $z(\mathbf{x})$ is field observation, $\eta(\cdot, \cdot)$ a Gaussian process with unknown parameters representing a computer model taking two arguments: the first argument is the independent variable $\mathbf{x}$, and the second argument a vector of parameters with θ being the true but unknown value of the parameters. The $\delta(\mathbf{x})$ part is a model inadequacy term, also a Gaussian process with unknown parameters. The last term is an observational iid error with $\epsilon \sim N(0, \lambda^2)$.

We will take reality to be

$$x + y^2 + y + xy + CE \tag{2}$$

where CE is correlated error. Observations are reality plus uncorrelated $N(0, \lambda^2)$ error. The model is

$$Ax + By^2 + CE \tag{3}$$

Here the A and B are parameters whose true but unknown values can be determined by comparing equation 2 with equation 3. The values of A and B are thus 1 and 1 respectively; write $\theta = (A, B)$ to denote the parameters collectively.

The model inadequacy term is thus

$$y + xy + CE$$

Note that this requires $A = B = 1$. We will take the model basis functions to be

$$h_1(\mathbf{x}, \theta) = (1, x, A, Ax, By^2)^T \tag{4}$$

Notes

- The model output will be
$$h_1(x, \theta)^T \beta_1 + CE$$
where $\beta_1 = (0, 0, 0, 1, 1)^T$ is the true value of the coefficients; see section 7 for this in use.
- The basis functions could be much more complicated than that or, indeed, much simpler. We could have chosen, for example, $h_1(x, \theta) = (1, x, y, x^2, y^2, A, B, Ax, Ay, Bx, By, Ax^2, Ay^2, Bx^2, By^2)^T$; or, going the other way, $h_1(x, \theta) = 1$.

The model inadequacy term then has a mean of $y + xy$. The basis functions for the model inadequacy is then

$$h_2(x) = (y, xy)^T \tag{5}$$

and the model inadequacy then has a mean of

$$h_2(x)\beta_2$$

where $\beta_2 = (1, 1)^T$. Again note the great scope for choosing basis functions (although in the case of $h_2()$ there are no parameters to consider).

4.1 Defining regressor functions

We need an `h1.int()`:

```
> h1.int <- function(xin) {
+     out <- c(1, xin[1], xin[3], xin[1] * xin[3], xin[2]^2 * xin[4])
+     names(out) <- c("const", "x", "A", "Ax", "By.sq")
+     return(out)
+ }
```

Note how the argument `xin` is just `c(x,theta)`, where `x` is the independent variables and `theta` the model parameters. So `xin[1]` is x, `xin[2]` is y, `xin[3]` is A, and `xin[4]` is B. See how function `h1.int()` creates the basis functions $(1, x, A, Ax, By^2)^T$, as per equation 4.

Now we need a vectorized version:

```
> H1.int <- function(D1) {
+     if (is.vector(D1)) {
+         D1 <- t(D1)
+     }
+     out <- t(apply(D1, 1, h1.int))
+     colnames(out) <- c("const", "x", "A", "Ax", "By.sq")
+     return(out)
+ }
```

Notes

- Function `H1.int` is the one needed as arguments to functions like `stage1()`; function `h1.int()` is not needed except from within `H1.int()`.

- See how the columnnames are specified. Again, not strictly necessary but strongly recommended
- See the special consideration for vector `D1`

Similarly for the model inadequacy function:

```
> h2.int <- function(x) {
+     out <- c(x[1], x[1] * x[2])
+     names(out) <- c("h2.x", "h2.xy")
+     return(out)
+ }
> H2.int <- function(D2) {
+     if (is.vector(D2)) {
+         D2 <- t(D2)
+     }
+     out <- t(apply(D2, 1, h2.int))
+     colnames(out) <- names(h2.int(D2[1, , drop = TRUE]))
+     return(out)
+ }
```

5 Hyperparameter object

We now need a function `phi.fun.int()` to create an appropriate hyperparameter object. Note that function `phi.change()` is generic (so we don't need to write one). The best way to write `phi.fun.int()` is to proceed by analogy and modify function `phi.fun.toy()`, line by line. Most of the function is straightforward to create. Note that the only parts that need changing in this case are the internal functions `pdm.maker.psi1()` and `pdm.maker.psi1()`; I use "pdm" to denote "positive definite matrix". These two functions create positive-definite matrices $\mathbf{\Omega}_x$ and $\mathbf{\Omega}_t$ that are used to determine $c_1\left((x,t),(x',t')\right)$. These are defined by:

```
> pdm.maker.psi1 <- function(psi1) {
+     jj.omega_x <- diag(psi1[1:2])
+     rownames(jj.omega_x) <- names(psi1[1:2])
+     colnames(jj.omega_x) <- names(psi1[1:2])
+     jj.omega_t <- diag(psi1[3:4], ncol = 2)
+     rownames(jj.omega_t) <- names(psi1[3:4])
+     colnames(jj.omega_t) <- names(psi1[3:4])
+     sigma1squared <- psi1[5]
+     return(list(omega_x = jj.omega_x, omega_t = jj.omega_t, sigma1squared = sigma1squared))
+ }
> pdm.maker.psi2 <- function(psi2) {
+     jj.omegastar_x <- diag(psi2[1:2], ncol = 2)
+     sigma2squared <- psi2[3]
+     return(list(omegastar_x = jj.omegastar_x, sigma2squared = sigma2squared))
+ }
```

Notes

- In function `phi.fun.toy()` there are two functions with the same name. They extract from vectors `psi1` and `psi2` a positive definite matrix used in $c_1(\cdot,\cdot)$ and $c_2((\cdot,\cdot),(\cdot,\cdot))$.
- The purpose of all this is to implement the fact that $\mathbf{\Omega}_x$ and $\mathbf{\Omega}_t$ are *arbitrary* functions of `psi1`. Here, it's dead easy because the matrices are diagonal and the values just correspond to consecutive elements of `psi1` but in general there may be nonzero off-diagonal elements of $\mathbf{\Omega}_x$ which may be a complicated function of `psi1`.

Given this, we can now change `phi.int`, using function `phi.change()`. Note that this function is generic, so we do not need to write another version. The best way to call `phi.change()` is to define intermediate variables that begin with `jj`:

```
> jj.psi1 <- 1:5
> names(jj.psi1) <- c("x", "y", "A", "B", "s1sq")
> jj.psi2 <- 1:3
> names(jj.psi2) <- c("x", "y", "s1sq")
> jj.mean1 <- rep(1, 5)
> names(jj.mean1) <- names(jj.psi1)
> jj.sigma1 <- diag(c(1.1, 1.1, 1.2, 1.3, 1.1))
> rownames(jj.sigma1) <- names(jj.psi1)
> colnames(jj.sigma1) <- names(jj.psi1)
> jj.mean2 <- c(1, 0.1, rep(1.1, 3))
> names(jj.mean2) <- c("rho", "lambda", names(jj.psi2))
> jj.sigma2 <- diag(c(1, 0.2, 1.1, 1.1, 1.2))/10
> rownames(jj.sigma2) <- names(jj.mean2)
> colnames(jj.sigma2) <- names(jj.mean2)
> jj.mean.th <- 1:2
> names(jj.mean.th) <- c("A", "B")
> jj.sigma.th <- diag(c(1.5, 1.7))
> rownames(jj.sigma.th) <- names(jj.mean.th)
> colnames(jj.sigma.th) <- names(jj.mean.th)
```

These variables may be passed directly to `phi.fun.int()`:

```
> phi.int <- phi.fun.int(rho = 1, lambda = 1, psi1 = jj.psi1, psi2 = jj.psi2,
+     psi1.apriori = list(mean = jj.mean1, sigma = jj.sigma1),
+     psi2.apriori = list(mean = jj.mean2, sigma = jj.sigma2),
+     theta.apriori = list(mean = jj.mean.th, sigma = jj.sigma.th),
+     power = 2)
```

Notes:

- The purpose of the call to `phi.fun.int()` was to create a hyperparameter object, here `phi.int`.
- you only have to call this function, with all its complicated arguments, once. Save the result in, say, `jj` and then use function `phi.change()`, which is generic, and much simpler to use.
- The value for rho and lambda and power are just scalars
- `psi1` and `psi2` are structures, or named vectors (it's always good practice to elements: two for x and y, two for the parameters A and B, and one for sigma1squared. Vector psi2 has three: two for x and y, and one for `sigma2squared`.
- the priors are as expected. Remember that the apriori distributions are over psi1 or psi2, not just the roughness lengths.

So now we have a working hyperparameter object `phi.int`, we can modify it with generic function `phi.change()`:

```
> phi.int2 <- phi.change(old.phi = phi.int, phi.fun = phi.fun.int,
+     rho = 3)
> print(phi.int2$rho)

[1] 3
```

Note that function `phi.change()` takes an `old.phi` argument, which is a working hyperparameter object to be modified. It also takes a `phi.fun` argument, which is the name of a hyperparameter creation function, in this case `phi.fun.int`.

6 Functions E.theta.int() and Edash.theta.int()

The online help page for `E.theta.toy()` discusses how to create a new function for use in a particular example.

We need to define functions `E.theta.int()` and `Edash.theta.int()`. These functions change when the basis functions `h1.int()` and `h2.int()` change, because of possible nonlinearity.

The function `E.theta.toy()` is relatively simple because there the basis functions were linear in θ, so *in this case* expectation commutes past taking the basis functions[1]

The first step is to examine function `E.theta.toy()`:

```
> E.theta.toy

function (D2 = NULL, H1 = NULL, x1 = NULL, x2 = NULL, phi, give.mean = TRUE)
{
    if (give.mean) {
        m_theta <- phi$theta.apriori$mean
        return(H1(D1.fun(D2, t.vec = m_theta)))
    }
    else {
        out <- matrix(0, 6, 6)
        out[4:6, 4:6] <- phi$theta.apriori$sigma
        return(out)
    }
}
<environment: namespace:calibrator>
```

The function is in two bits: one if `give.mean` is `TRUE`, and one for it being `FALSE`. If `TRUE`, it returns `H1()` using the input value of `D2` and $\overline{\theta}$ (as determined from the *a priori* distribution). If `FALSE`, it needs to return the matrix described in the help file[2] which in this case is independent of `x1` and `x2`. This matrix is 6-by-6 because it is the same size as the variance matrix, and that is 6-by-6 because $h(x) =$ `c(1,x)` [recall that here `x` is a single row of a dataframe like `D1.toy`—actually, it uses the output of `D1.fun()`—and this has five columns].

That was straightforward because the toy case included only linear basis functions. In the intermediate case presented here, the basis functions are nonlinear so expectation will not commute past `h1.int()`. Recall that function `h1.int()` includes a product:

```
> h1.int

function (xin)
{
    out <- c(1, xin[1], xin[3], xin[1] * xin[3], xin[2]^2 * xin[4])
    names(out) <- c("const", "x", "A", "Ax", "By.sq")
    return(out)
}
```

(the fourth element of the output, viz `xin[1]*xin[3]`, is nonlinear).

The fact that $E(X^2) \neq (E(X))^2$ complicates the matrix returned by function `E.theta.int()` (see `?E.theta.int` for details of this matrix). But we may copy from function `E.theta.toy()` and modify as necesary:

[1] This is not true in general. Consider a nonlinear case, for example $h_1(x,\theta) = (1, A^2x)^T$. Now $E_\theta\left(h_1(x,\theta)\right)^T$—that is, the expectation of $(1, A^2x)^T$ under the prior distribution for θ—will be $\left(1, (\overline{A}^2 + \sigma_A^2)x\right)^T$.

[2] $E_\theta(h \cdot h^T) - E_\theta(h) \cdot E_\theta(h)^T$

```
> E.theta.int <- function(D2 = NULL, H1 = NULL, x1 = NULL, x2 = NULL,
+     phi, give.mean = TRUE) {
+     if (give.mean) {
+         m_theta <- phi$theta.apriori$mean
+         return(H1(D1.fun(D2, t.vec = m_theta)))
+     }
+     else {
+         out <- matrix(0, 5, 5)
+         out[3, 3] <- phi$theta.apriori$sigma[1, 1]
+         out[3, 4] <- phi$theta.apriori$sigma[1, 1] * x1[1]
+         out[4, 3] <- phi$theta.apriori$sigma[1, 1] * x2[1]
+         out[4, 4] <- phi$theta.apriori$sigma[1, 1] * x1[1] *
+             x2[1]
+         out[5, 5] <- phi$theta.apriori$sigma[2, 2] * x1[2] *
+             x2[2]
+         return(out)
+     }
+ }
```

Notes

- The object returned when `give.mean` is `TRUE` does not need changing, because `h1.int()` is linear in `A` and `B`. If there were nonlinear terms there (such as A^2) we would need to add a variance.
- The online help for `E.theta.toy()` discusses the `give.mean` being `FALSE` part. Here, we use the fact that the prior distribution has zero correlation between A and B. The thing at the top of page 5 of the supplement should be

$$E_\theta\left(\mathbf{h}_1\left(\mathbf{x}_j,\boldsymbol{\theta}\right)\mathbf{h}_1\left(\mathbf{x}_j,\boldsymbol{\theta}\right)^T\right)=E_\theta\begin{pmatrix}1 & x & A & Ax & By^2\\ x & x^2 & Ax & Ax^2 & Bxy^2\\ A & Ax & A^2 & A^2x & ABy^2\\ Ax & Ax^2 & A^2x & A^2x^2 & ABxy^2\\ By^2 & Bxy^2 & ABy^2 & ABxy^2 & B^2y^4\end{pmatrix} \tag{6}$$

which has value

$$\begin{pmatrix}1 & x & \overline{A} & \overline{A}x & \overline{B}y^2\\ x & x^2 & \overline{A}x & \overline{A}x^2 & \overline{B}xy^2\\ \overline{A} & \overline{A}x & \overline{A}^2+\sigma_A^2 & \left(\overline{A}^2+\sigma_A^2\right)x & \left(\overline{A}\,\overline{B}+\mathrm{cov}(A,B)\right)y^2\\ \overline{A}x & \overline{A}x^2 & \left(\overline{A}^2+\sigma_A^2\right)x & \left(\overline{A}^2+\sigma_A^2\right)x^2 & \left(\overline{A}\,\overline{B}+\mathrm{cov}(A,B)\right)xy\\ \overline{B}y^2 & \overline{B}xy^2 & \left(\overline{A}\,\overline{B}+\mathrm{cov}(A,B)\right)y^2 & \left(\overline{A}\,\overline{B}+\mathrm{cov}(A,B)\right)xy^2 & \left(\overline{B}^2+\sigma_B^2\right)y^4\end{pmatrix}$$

where an overline denotes expectation with respect to θ, and σ_A^2 denotes variance (with respect to θ) of A. So the object to return (when `give.mean` is `FALSE`) is this, minus $E_\theta\left(h_1(x,\theta)\right)E_\theta\left(h_1(x,\theta)\right)^T$ as documented. This would be

$$\begin{pmatrix}0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0\\ 0 & 0 & \sigma_A^2 & \sigma_A^2x & \mathrm{cov}(A,B)y^2\\ 0 & 0 & \sigma_A^2x & \sigma_A^2x^2 & \mathrm{cov}(A,B)xy\\ 0 & 0 & \mathrm{cov}(A,B)y^2 & \mathrm{cov}(A,B)xy & \sigma_B^2y^4\end{pmatrix}$$

We can now use the fact that $E(XY)=E(X)\cdot E(Y)$ if X and Y are independent random variables. In this case, because the prior variance matrix (viz, `jj.psi1.apriori`) is diagonal,

the different terms are indeed independent. So the covariance is zero and the matrix reduces to

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \sigma_A^2 & \sigma_A^2 x & 0 \\ 0 & 0 & \sigma_A^2 x & \sigma_A^2 x^2 & 0 \\ 0 & 0 & 0 & 0 & \sigma_B^2 y^4 \end{pmatrix}$$

which is implemented in function `E.theta.int()` given above[3]

- There is probably a better way to return this matrix.

6.1 Function Edash.theta.int()

Function `Edash.theta.int()` returns expectation of $h_1(x,\theta)$ with respect to the normal distribution

$$\mathcal{N}\left((\boldsymbol{V}_\theta^{-1}+2\boldsymbol{\Omega}_t)^{-1}(\boldsymbol{V}_\theta^{-1}\boldsymbol{\Omega}_\theta+2\boldsymbol{\Omega}_t\mathbf{t}_k),(\boldsymbol{V}_\theta^{-1}+2\boldsymbol{\Omega}_t)^{-1}\right) \tag{7}$$

Fortunately, $E'_\theta(h\cdot h^T)$ is not required.

As the basis functions used are linear in theta[4], expectation WRT the dashed distribution commutes past the regressor function $h_1(\cdot,\cdot)$. So we can just use the toy function:

```
> Edash.theta.int <- Edash.theta.toy
```

7 Data

We now generate some data: code runs, and observations.

This is done in order to test the routines: by generating data with known parameters and hyperparameters we can verify that the package can reproduce at least approximately correct values.

Consider the following equation, taken from KOH:

$$z_i = \zeta(x_i)+e_i = \rho\eta(x_i,\theta)+\delta(x_i)+e_i \tag{8}$$

where z_i is the i-th observation, $\zeta(x_i)$ is the true value at point x_i, ρ a calibration factor (of notional value 1), $\eta(\cdot,\cdot)$ the code viewed as a function of observation point and parameter value, θ

[3]Returning to the nonlinear example given in the previous footnote, viz $h=h_1(x,\theta)=(1,A^2x)^T$. This would have $E_\theta(h)=\left(1,x(\overline{A}^2+\sigma_A^2)\right)^T$ and $E_\theta(h)\cdot E_\theta(h)^T=\begin{pmatrix} 1 & x\left(\overline{A}^2+\sigma_A^2\right) \\ x\left(\overline{A}^2+\sigma_A^2\right) & x^2\left(\overline{A}^2+\sigma_A^2\right)^2 \end{pmatrix}$.

But what is needed by KOH is

$$E_\theta\left(h\cdot h^T\right)=E_\theta\begin{pmatrix} 1 & A^2x \\ A^2x & A^4x^2 \end{pmatrix}=\begin{pmatrix} 1 & x\left(\overline{A}^2+\sigma_A^2\right) \\ x\left(\overline{A}^2+\sigma_A^2\right) & x^2\left(\overline{A}^4+6\overline{A}^2\sigma_A^2+3(\sigma_A^2)^2\right) \end{pmatrix}.$$

The manpage for function `E.theta.toy()` says how, if argument `give.mean` is `FALSE`, the value returned should be the thing that has to be added to $E_\theta(h)\cdot E_\theta(h)^T$ to give $E_\theta\left(h\cdot h^T\right)$. The motivation for considering it this way is that `E.theta.toy(...,give=FALSE)` will return a zero matrix for basis functions linear in θ.

So, for this $h(\cdot,\cdot)$, function `E.theta.int()` would have to return the difference between $E_\theta\left(h\cdot h^T\right)$ and $E_\theta(h)\cdot E_\theta(h)^T$, which would be $\begin{pmatrix} 0 & 0 \\ 0 & x^2\left(4\overline{A}^2\sigma_A^2+2(\sigma_A^2)^2\right) \end{pmatrix}$.

[4]Returning to our nonlinear example, footnotes passim, viz $h_1(x,\theta)=(1,xA^2)^T$, we can see from equation 7 that $E'\theta=\left(1,x\left(\mu'^2_A+\sigma'^2_A\right)\right)^T$ where μ'_A is the mean of the distribution in equation 7, that is, the first element of the vector $(\boldsymbol{V}_\theta^{-1}+2\boldsymbol{\Omega}_t)^{-1}(\boldsymbol{V}_\theta^{-1}\boldsymbol{\Omega}_\theta+2\boldsymbol{\Omega}_t\mathbf{t}_k)$, and σ'^2_A is the variance—that is the top left element of $(\boldsymbol{V}_\theta^{-1}+2\boldsymbol{\Omega}_t^{-1})$ (NB: this argument is true whether or not the variance matrix $\boldsymbol{V}_\theta$ and the positive definite scales matrix $\boldsymbol{\Omega}_t$ are diagonal)

the true but unknown set of parameters, and $\delta(\cdot)$ the model inadequacy term, and $e_i \sim \mathcal{N}(0, \lambda^2)$ is an observational error term.

So, do the model first. Recall that the model is a Gaussian process and we have discussed the mean in section 4. The variance matrix is given by function `corr.matrix()` of package `emulator`. We know, *ex cathedra,* that $A = B = 1$:

```
> theta.TRUE <- c(1, 1)
```

and we can specify beta1 and psi1:

```
> beta1.TRUE <- c(0, 0, 0, 1, 1)
> psi1.TRUE <- c(4, 4, 4, 4, 0.5)
```

Now we need to create a design matrix:

```
> two.designs <- rbind(D1.int, D1.fun(x.star = D2.int, t.vec = theta.TRUE))
```

Here, `two.designs` is a design matrix of `n1+n2` rows; the first `n1` rows of which are the code observation points, and the last `n2` rows are the field observation points but with the true parameter value added.

Just have a look at the first three and last three lines:

```
> two.designs[c(1:3, (n1 + n2):(n1 + n2 - 2)), ]
```

```
                  x         y     A     B
coderun.1 0.0750000 0.9250000 0.825 0.975
coderun.2 0.0250000 0.1750000 0.225 0.575
coderun.3 0.7750000 0.0750000 0.475 0.325
obs.21    0.5476190 0.7857143 1.000 1.000
obs.20    0.4523810 0.9761905 1.000 1.000
obs.19    0.8333333 0.8809524 1.000 1.000
```

See how the block of `1.0`s in the lower right corner corresponds to appending the true value of θ to the design matrix.

The next step is to sample from the appropriate multivariate Gaussian distribution:

```
> jj.mean <- H1.int(two.designs) %*% beta1.TRUE
> jj.sigma <- psi1.TRUE[5] * corr.matrix(two.designs, scales = psi1.TRUE[1:4])
> code.and.obs <- as.vector(rmvnorm(n = 1, mean = jj.mean, sigma = jj.sigma))
> y.int <- code.and.obs[1:n1]
> z.int <- code.and.obs[(n1 + 1):(n1 + n2)]
> names(y.int) <- rownames(D1.int)
> head(y.int)
```

```
 coderun.1  coderun.2  coderun.3  coderun.4  coderun.5  coderun.6
 0.8918551  0.2600716 -0.7921402  1.1085165 -0.1104723 -0.3314009
```

Notes

- The mean, `jj.mean`, is given by the linear combination of the regressor basis as discussed in section 4
- The model η is a Gaussian process. The mean is specified, the variance matrix given by the correlation function `corr.matrix()` of package `emulator`. The value for σ_1^2 is one (ie the fifth element of `psi1.TRUE`).
- Function `rmvnorm()` returns a matrix, which has to be converted to a vector.
- The names of `y.int` have to be specified explicitly.
- Observe how `n=1` in the call to `rmvnorm()`. We are making a *single* observation of a multivariate Gaussian distribution.

7.1 Observations and model inadequacy

To create reality, we have to generate model observations using the true but unknown parameter values.

We now have to add the model inadequacy term to `z.int`. First, specify the parameters and hyperparameters of the model inadequacy:

```
> beta2.TRUE <- c(1, 1)
> psi2.TRUE <- c(3, 3, 0.6)
```

Which give the true values. Now create model inadequacy:

```
> jj.mean <- drop(H2.int(D2.int) %*% beta2.TRUE)
> jj.sigma <- corr.matrix(D2.int, scales = psi2.TRUE[1:2]) * psi2.TRUE[3]
> model.inadequacy <- rmvnorm(n = 1, mean = jj.mean, sigma = jj.sigma)
> z.int <- as.vector(z.int + model.inadequacy)
> names(z.int) <- rownames(D2.int)
```

Notes

- The overall purpose of the above code fragment is to create observations drawn from the appropriate multivariate Gaussian distribution. The key is the fourth line, in which the model observations `cond.gp` have model inadequacy added.
- Model inadequacy has two components: the mean (`jj.mean`), generated using the true coefficients and the basis functions; and the correlated residual.
- The correlated error is added using `rmvnorm()` with a variance matrix generated by `corr.matrix()`.
- The correlated error is independent of the model runs, and in particular is not a function of θ.

Now add observational error:

```
> lambda.TRUE <- 0
> jj.obs.error <- rnorm(n2) * lambda.TRUE
> z.int <- z.int + jj.obs.error
> head(z.int)
```

```
    obs.1     obs.2     obs.3     obs.4     obs.5     obs.6
 2.154719  2.164851  2.253942  2.414639  2.084959 -1.202877
```

Notes

- The observation errors are uncorrelated Gaussian with a mean of 0 and a variance of 0.1.
- Thus λ is 0.1

And finally we need to create a full data vector `d.int`

```
> d.int <- c(y.int, z.int)
```

8 Intermediate results

We now use some functions that are part of the `BACCO` bundle.

Note that the numbers given above will vary from instantiation to instantiation.

OK, so let's change the hyperparameter object to contain the true values:

```
> phi.true <- phi.change(phi.fun = phi.fun.int, old.phi = phi.int,
+     psi1 = psi1.TRUE, psi2 = psi2.TRUE, lambda = 0.1, rho = 1)
```

and then use these hyperparameters to estimate the coefficients:

```
> betahat.fun.koh(theta = theta.TRUE, d = d.int, D1 = D1.int, D2 = D2.int,
+     H1 = H1.int, H2 = H2.int, phi = phi.true)
```

```
            [,1]
const  0.1334844
x     -0.7251545
A     -0.9471333
Ax     1.3292263
By.sq  2.6285464
h2.x   2.7006887
h2.xy -0.3122296
```

(note the last argument to `betahat.fun.koh()`).

We know the correct answer should be `c(beta1.TRUE,beta2.TRUE)=c(0,0,0,1,1,1,1)`, so there is evidently a lot of scatter. Try redefining `psi1.TRUE` and `psi2.TRUE` so that the variance (ie the last element of each vector) is smaller. Or for that matter, using a larger value of `n1` in section 3.

Just as a point of interest, we will estimate the betas but using the wrong parameters (recall that in practice, the parameters' true value is not known).

```
> betahat.fun.koh(theta = c(-5, 5), d = d.int, D1 = D1.int, D2 = D2.int,
+     H1 = H1.int, H2 = H2.int, phi = phi.true)
```

```
             [,1]
const -0.03504270
x      0.23946747
A      0.36320726
Ax    -0.93212857
By.sq  0.96135648
h2.x   0.77499208
h2.xy -4.00875507
```

This is far from the true values because we have used a very inaccurate value for theta.

8.1 Calibration

As a bit of fun, we can use equation 8 of the supplement, implemented in BACCO by `p.eqn8.supp()`. This equation gives the probability of theta, given d and psi. Note that this formula is conditional on the correct hyperparameters psi, so we will use `phi.true`.

```
> p.eqn8.supp(theta = c(1, 1), D1 = D1.int, D2 = D2.int, H1 = H1.int,
+     H2 = H2.int, d = d.int, phi = phi.true)
```

```
[1] 3253170
```

(Recall that this function gives a number *proportional* to the true probability). Now try a different value for theta:

```
> p.eqn8.supp(theta = c(5, -6), D1 = D1.int, D2 = D2.int, H1 = H1.int,
+     H2 = H2.int, d = d.int, phi = phi.true)
```

```
[1] 712.1264
```

See how this is lower, because the value of theta is unlikely to be the true one. So we could use a maximum likelihood estimator (which would have to use numerical optimization techniques) to estimate theta, if we knew the correct values to use for the hyperparameters.

In practice, the hyperparameters are not known in advance. Estimating them is the subject of the next section.

9 Estimating the hyperparameters

The BACCO bundle includes two functions to automate the determination of the hyperparameters: `stage1()` and `stage2()`. The bundle also includes a `stage3()` function which gives a MLE for theta (but no such stage3 appears in KOH).

The function calls in this section are very computationally intensive and the calls include a number of timesaving features (such as very short optimization) that degrade the quality of the predictions.

These features are discussed where appropriate but the user is advised to fiddle with them to get better results. YMMV.

9.1 Stage 1

KOH proposed estimating the hyperparameters in two stages. Stage 1 used just the code output data `y` to estimate `psi1`. In the `calibrator` bundle, this is accomplished by `stage1()`:

```
> phi.stage1 <- stage1(D1 = D1.int, y = y.int, H1 = H1.int, maxit = 10,
+     method = "SANN", trace = 0, do.print = FALSE, phi.fun = phi.fun.int,
+     phi = phi.int)
```

Notes

- Function `stage1()` returns a hyperparameter object, which contains optimized value for `psi1`.
- The method used is simulated annealing, because with `maxit=1` it finishes very quickly.
- The function takes a hyperparameter object, in this case `phi.int`.
- By default, function `stage1()` maximizes the posterior probability, so the prior (which is part of the hyperparameter object) makes a difference.

We can examine the output of `stage1()` directly:

```
> phi.stage1$psi1

       x        y        A        B     s1sq
3.267233 1.725379 1.511408 2.784563 1.405226
```

Recall that `phi.stage1$psi1` is a vector of free parameters that are used to calculate $c_1(\mathbf{x}, \mathbf{x}')$.

Further recall that $c_1\left((\mathbf{x},\mathbf{t}),(\mathbf{x}',\mathbf{t}')\right) = \sigma_1^2 \exp\left\{-(\mathbf{x}-\mathbf{x}')^T\mathbf{\Omega}_x(\mathbf{x}-\mathbf{x}') - (\mathbf{t}-\mathbf{t}')^T\mathbf{\Omega}_t(\mathbf{t}-\mathbf{t}')\right\}$ where σ_1^2, and $\mathbf{\Omega}_x$ and $\mathbf{\Omega}_t$ are positive definite matrices that are functions of `psi1`.

In this case we have

```
> phi.stage1$sigma1squared

    s1sq
1.405226

> phi.stage1$omega_x

         x        y
x 3.267233 0.000000
y 0.000000 1.725379

> phi.stage1$omega_t

         A        B
A 1.511408 0.000000
B 0.000000 2.784563
```

9.2 Stage 2

Stage 2 is the estimation of ρ, λ, and `psi2`. This is accomplished by function `stage()`.

NB: stage 2 is very computationally intensive!

OK, use `stage2()`:

```
> use1 <- 1:10
> use2 <- 1:11
> phi.stage2 <- stage2(D1 = D1.int[use1, ], D2 = D2.int[use2, ],
+     H1 = H1.int, H2 = H2.int, y = y.int[use1], z = z.int[use2],
+     extractor = extractor.int, phi.fun = phi.fun.int, E.theta = E.theta.int,
+     Edash.theta = Edash.theta.int, maxit = 1, method = "SANN",
+     phi = phi.stage1)

                            x         y      s1sq
0.0000000 0.0000000 0.0000000 0.6931472 1.0986123
[1] -39.60726
sann objective function values
initial       value 39.607255
final         value 39.607255
sann stopped after 0 iterations
```

Notes

- This function takes a long long long time to run, and is very computationally slow.
- The function uses only the first 10 code runs and the first 11 field observations. You can change this by modifying `use1` and `use2` in the above chunk.
- The above function call is an absolute minimum working model. It uses SANN with only *one* function evaluation.
- The start point is the output from `stage1()`, viz `phi.stage1`.
- The purpose of function `stage2()` is to optimize the posterior probability of the hyperparameters rho, lambda, sigma2squared, and the lengthscales for $c_2(\cdot, \cdot)$.

The modified hyperparameters are

```
> phi.stage2$rho

1

> phi.stage2$lambda

1

> phi.stage2$sigma2squared

s1sq
   3

> phi.stage2$psi2

   x    y s1sq
   1    2    3
```

(although, given the extremely short optimization run above, these parameters may well not be different from the original).

10 Calibrated prediction

Function `EK.eqn10.supp()` carries out calibrated prediction as per section 4.2 of KOH2. It should come as no surprise that

- The R code is complicated and impenetrable
- The function arguments are tedious, error-prone, and complicated
- The mathematics are complicated and impenetrable
- The numerics take an interminably long time to complete
- Even with the simplest case possible, accurate results require more computing power than is available in the entire universe.

... but, it *is* implemented.

The first step is to read `help(EK.eqn10.supp)`, which gives a working example.

There are two arguments that are not covered above. The first is `X.dist` and the second is `hbar`. These are discussed in the online help pages.

For `X.dist`, we need to define an uncertainty distribution on the independent variables. Working from `X.dist.toy`, we may copy it directly:

```
> jj.xdist.mean <- rep(0.5, 2)
> names(jj.xdist.mean) <- c("x", "y")
> jj.xdist.var <- 0.05 + diag(c(0.1, 0.1))
> rownames(jj.xdist.var) <- c("x", "y")
> colnames(jj.xdist.var) <- c("x", "y")
> X.dist.int <- list(mean = jj.xdist.mean, var = jj.xdist.var)
```

thus defining a Gaussian uncertainty distribution for `X`. We now need a function `hbar`. The manpage discusses this and gives an example. Note that the example is excruciatingly simple because there the basis functions are linear in `x`. And in the intermediate case they are not.

To write a suitable function, we need to remind ourselves what the basis functions `h1.int()` and `h2.int()` are. Look back at equations 4 and 5.

Take the top bit first. This is $\rho E_X\{h_1(\mathbf{x},\boldsymbol{\theta})\}$, that is, expectation with respect to `X.dist`. Recall that $h_1(\mathbf{x},\boldsymbol{\theta}) = (1, x, A, Ax, By^2)^T$, so the top bit is

$$\rho E_X\left(1, x, A, Ax, By^2\right)^T = \rho\left(1, \overline{x}, A, A\overline{x}, B\left(\overline{y}^2 + \sigma_y^2\right)\right)^T \tag{9}$$

where the overline refers to expectation with respect to `X.dist`. The bottom bit is $E_X\{h_2(\mathbf{x})\}$, which is (recall that $h_2(\mathbf{x}) = (x, xy)^T$):

$$E_X\,(x, xy)^T = \left(\overline{x}, \overline{x}\cdot\overline{y} + \mathrm{cov}(x, y)\right)^T \tag{10}$$

and we can use the fact that `X.dist.int` specifies zero correlation between the two independent variables' uncertainty distribution to reduce this to

$$\left(\overline{x}, \overline{x}\cdot\overline{y}\right)^T$$

OK, using the toy example as a guide, we can define a working `hbar.fun.int()`:

```
> hbar.fun.int <- function(theta, X.dist, phi) {
+     if (is.vector(theta)) {
+         theta <- t(theta)
+     }
+     first.bit <- phi$rho * H1.int(D1.fun(X.dist$mean, theta))
+     first.bit[, 5] <- first.bit[, 5] + theta[, 2] * X.dist$var[2,
```

```
+         2]
+     second.bit <- H2.int(X.dist$mean)
+     jj.names <- colnames(second.bit)
+     second.bit <- kronecker(second.bit, rep(1, nrow(first.bit)))
+     colnames(second.bit) <- jj.names
+     return(t(cbind(first.bit, second.bit)))
+ }
```

notes

- See how the fifth column of `first.bit` has to have a component for the variance added, to match the variance term in equation 9.
- There is no such addition to `second.bit` because the random variables in `X.dist.int` have zero correlation.
- Everything else just follows from `hbar.fun.toy()`.

Now we can conduct a calibrated prediction:

```
> jj <- EK.eqn10.supp(X.dist = X.dist.int, D1 = D1.int, D2 = D2.int,
+     H1 = H1.int, H2 = H2.int, d = d.int, hbar.fun = hbar.fun.int,
+     lower.theta = c(-3, -3), upper.theta = c(3, 3), extractor = extractor.int,
+     phi = phi.stage2, eps = 0.8)
> jj

[1] 1.824533
```

Notes

- We use the optimized value for `phi`, that is `phi.stage2` (not the true value).
- I have set `eps` to 0.8, as `adapt()` is very slow with the default settings.

11 Metropolis-Hastings

The Metropolis-Hastings method allows one to sample from a PDF that is only specified up to a constant. It is useful for posterior PDFs of the type given by equation 8 of the supplent (`p.eqn8.supp()` in the package: only a likelihood is given.

See the manpage for `MH.Rd` for examples of the Metropolis-Hastings method in use.

For ease of use, we will define a wrapper:

```
> p <- function(theta) {
+     p.eqn8.supp(theta = theta, D1 = D1.int, D2 = D2.int, H1 = H1.int,
+         H2 = H2.int, d = d.int, phi = phi.true)
+ }
> p(c(1, 1))

[1] 3253170

> p(c(10, 10))

[1] 3336.583
```

So, let's sample from it:

```
> ss <- MH(n = 10, start = c(1, 1), sigma = diag(2), pi = p)
```

Figure 1 shows a sample of size 10 from the posterior.

```
> plot(jitter(ss), xlab = "A", ylab = "B", main = "Sample from posterior: try with more points")
```

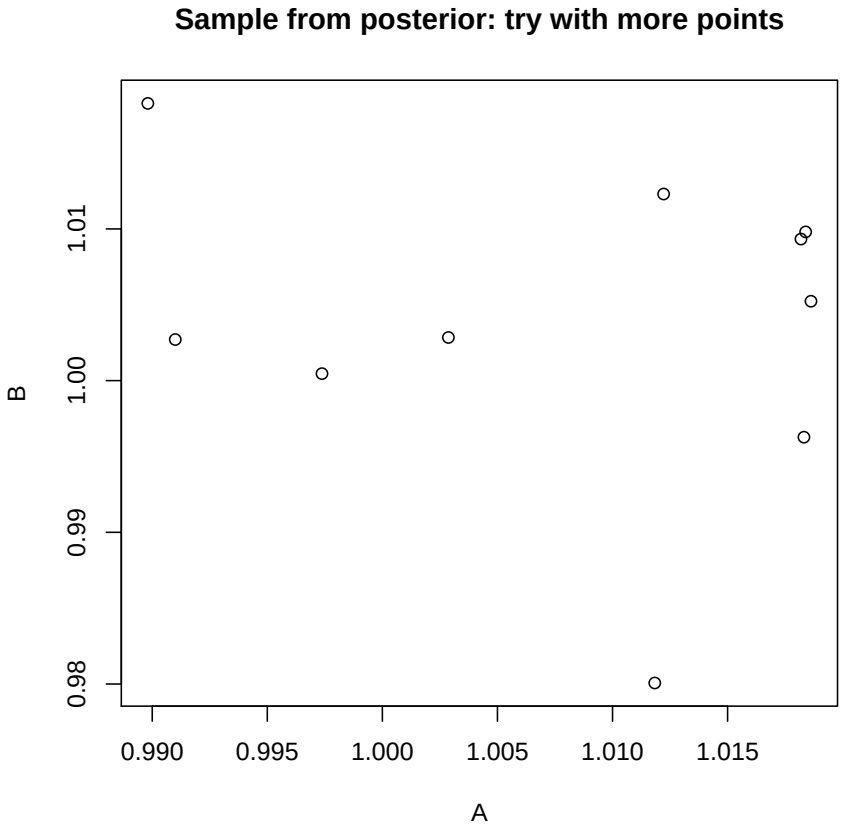

Figure 1: Sample from the posterior distribution of theta, made using the Metropolis-Hastings method

Robin K. S. Hankin
National Oceanography Centre, Southampton
European Way
Southampton SO14 3ZH
United Kingdom
E-mail: r.hankin@noc.soton.ac.uk URL: http://www.noc.soton.ac.uk